BASIC

FLUKE®

BASIC

FLUKE®

Digitized by the Internet Archive
in 2025

https://archive.org/details/manualsplus_08251

BASIC

P/N 716639
January 1984 Rev 1 5/85
©1985 John Fluke Mfg. Co., Inc
All rights reserved. Litho in U.S.A.

FLUKE®

Contents

1	**HOW TO USE THIS MANUAL**	**1-1**
	Introduction	1-3
	BASIC Manual	1-5
	BASIC Reference Manual	1-6
	How to Read Syntax Diagrams	1-7
	Notation Conventions	1-9
2	**BASIC CONVENTIONS**	**2-1**
	Introduction	2-3
	Overview	2-3
	Exceptions to ANSI Standard BASIC	2-4
	Running the BASIC Interpreter Program	2-5
	Exiting BASIC	2-6
	BASIC Operating Modes	2-7
	Immediate Mode	2-7
	Edit Mode	2-9
	Run Mode	2-9
	Step Mode	2-10
	File Names	2-11
3	**SYSTEM FUNCTIONS**	**3-1**
	Introduction	3-3
	Overview	3-3
	System Functions for File Management	3-4
	COPY Statement	3-5
	DIR Statement	3-5
	EDIR Statement	3-5
	KILL Statement	3-6
	QDIR Statement	3-6
	RENAME	3-6

(continued on page ii)

Device Management Functions 3-7
ASSIGN Statement 3-7
PACK Statement 3-7
PROTECT Statement 3-7
UNPROTECT Statement 3-8

4 WRITING AND EDITING A PROGRAM 4-1
Introduction 4-3
Overview 4-3
Special Function Keys 4-3
Delete Key 4-3
PAGE MODE Switch and NEXT PAGE Keys 4-3
⟨CTRL⟩/Key-Modifier 4-4
Writing a Program 4-6
Entering a Program 4-7
Entering a Program from the Immediate Mode 4-7
LIST Command 4-9
DELETE Command 4-10
REN Command 4-11
Changing the Sequence of Program Lines 4-14
Entering a Program From the Edit Mode 4-15
Entering a Program Using the System Editor Program (EDIT.FD2) 4-15
Editing a Program 4-16
The BASIC Editor 4-17
Edit Mode Keys 4-19
Additional Editor Features 4-22

5 STORING AND RUNNING A PROGRAM 5-1
Introduction 5-3
Overview 5-3
Saving a Program 5-4
SAVE, RESAVE, SAVEL, and RESAVEL Immediate Mode Commands 5-4
Running a Program 5-6
OLD Immediate Mode Command 5-6
RUN Immediate Mode Command 5-8

6 DATA, OPERATORS, AND EXPRESSIONS 6-1
Introduction 6-3
Overview 6-3
Data Types 6-4
Floating-Point Data 6-4

(continued on page iii)

Floating-Point Constants 6-5
Integer Data 6-5
Integer Constants 6-6
String Data 6-6
String Constants 6-7
System Constants 6-7
BASIC Variables 6-8
Introduction To Variables 6-8
Floating-Point Variables 6-8
Integer Variables 6-8
String Variables 6-8
Array Variables 6-9
System Variables 6-11
Operators and Expressions 6-12
Assignment Statement: = ... 6-12
Arithmetic Operators: + - * / 6-12
Relational Operators: = < > <> <= >= 6-14
Numeric Comparisons 6-15
String Concatenation Operator: + ... 6-15
Logical (Bitwise) Operators 6-16
Binary Numbers 6-16
Twos Complement Binary Numbers 6-18
AND Operator 6-18
OR Operator 6-19
XOR Operator 6-19
NOT Operator 6-20
Operator Hierarchy 6-20
Use of Parentheses in Expressions 6-22

7 DATA STORAGE 7-1
Introduction 7-3
Overview 7-3
Using Arrays for Data Storage 7-4
Array Types and Differences 7-5
Advantages and Disadvantages 7-6
I/O Channels 7-7
Channel Input and Output 7-7
OPENing an Output Channel 7-8
OPENing an Input Channel 7-8
Close a Channel 7-8
Specifying File SIZE 7-9
Sequential Data Files 7-10
Creating a Sequential File 7-10
Reading a Sequential File 7-11

(continued on page iv)

Creating Main Memory Arrays 7-13
Using Main Memory Arrays 7-14
Using Main Memory Arrays as Ordinary Variables 7-14
Programming Techniques 7-15
Two Dimensional Arrays 7-16
Multiple Arrays 7-17
Redimensioning 7-17
Main Memory Arrays and Program Chaining 7-18
Serial Storage of Main Memory Arrays in Mass Storage . 7-19
Device Storage Size Requirements 7-20
Device File Size Calculation 7-20
Disk Size Calculation Example 7-21
Virtual Array Files 7-22
Definition 7-24
Advantages and Disadvantages 7-24
File Structure 7-26
Analogy 7-26
Virtual Array File Organization 7-27
Storage Size Requirements 7-29
Virtual Array Size Calculation 7-30
Creating Virtual Arrays 7-32
Using Virtual Arrays 7-33
Using Virtual Arrays as Ordinary Variables 7-33
Using Virtual Array Strings 7-34
Programming Techniques 7-35
Array Element Access 7-35
Splitting Arrays Among Files 7-36
Reusing Virtual Array Declarations 7-37
Equivalent Virtual Arrays 7-38

8 FUNCTIONS 8-1
Introduction 8-3
Overview 8-3
General Purpose Mathematical Functions 8-4
ABS() Function 8-5
ASH() Function 8-5
EXP() Function 8-5
INT() Function 8-5
LN() Function 8-6
LOG() Function 8-6
LSH() Function 8-6
MOD() Function 8-6
RND Function 8-7

(continued on page v)

SGN() Function 8-7
SQR() Function 8-7
Trigonometric Functions 8-8
ATN() Function 8-9
COS() Function 8-9
SIN() Function 8-9
TAN() Function 8-9
String Functions 8-10
ASCII() Function 8-11
CHR$() Function 8-11
CPOS() Function 8-11
DUPL$() Function 8-11
INSTR() Function 8-12
LCASE$() Function 8-12
LEFT() Function 8-12
LEN() Function 8-12
MID() Function 8-12
NUM$() Function 8-13
RAD$() Function 8-13
RIGHT() Function 8-13
SPACE$() Function 8-13
TAB() Function 8-13
UCASE$() Function 8-14
VAL() Function 8-14
User-Defined Functions 8-14

9 IEEE-488 Bus Input and Output Statements 9-1
Introduction 9-3
Overview 9-3
IEEE-488 Addressing 9-4
Device Addressing 9-5
Addressing Serial IEEE-488 Devices 9-7
Port Addressing 9-8
Initialization and Control Statements 9-9
CLEAR Statement 9-10
INIT Statement 9-10
LOCAL Statement 9-11
LOCKOUT Statement 9-11
ON PORT Statement 9-12
OFF PORT Statement 9-12
PASSCONTROL Statement 9-12
REMOTE Statement 9-12
SET SRQ Statement 9-13

(continued on page vi)

TRIG Statement 9-13
TERM Statement 9-13
TIMEOUT Statement 9-13
Input and Output Statements 9-14
INPUT Statement 9-14
INPUT LINE Statement 9-15
INPUT WBYTE Statement 9-15
INPUT LINE WBYTE Statement 9-15
PRINT and PRINT USING Statements 9-16
IEEE-488 Data Transfer Statements 9-17
RBYTE Statement 9-18
WBYTE Statement or Clause 9-18
RBYTE WBYTE Statement 9-18
RBIN Statement 9-19
RBIN WBYTE Statement 9-19
WBIN Statement 9-20
IEEE-488 Polling Statements 9-20
CONFIG Statement 9-21
ON SRQ and OFF SRQ Statements 9-21
ON PPOL and OFF PPOL Statements 9-21
PORTSTATUS() Function 9-22
PPL() Function 9-22
SPL Function 9-22
WAIT Statement 9-23

10 **RS-232 SERIAL INPUT AND OUTPUT** **10-1**
Introduction 10-3
Overview 10-3
RS-232-C Defined 10-4
Device and Port Addressing 10-5
Initialization 10-6
I/O Channels 10-6
OPENing a Serial Communication Channel 10-7
CLOSEing a Serial Communications Channel 10-8
Output and Input 10-9
PRINT Statement 10-9
INPUT Statement 10-10
INCOUNT() Function 10-12
INCHAR() Function 10-13
Sending BREAK 10-14
Establishing Serial Communications 10-15

(continued on page vii)

11 INTERRUPT PROCESSING 11-1

Introduction 11-3
Overview 11-3
Types of Interrupts 11-4
Error Interrupt 11-4
CTRL /C Interrupt 11-4
#n Interrupt 11-5
KEY Interrupt 11-5
PORT Interrupt 11-5
PPORT Interrupt 11-5
SRQ Interrupt 11-5
PPOL Interrupt 11-5
CLOCK Interrupt 11-5
INTERVAL Interrupt 11-5
Hierarchy of Interrupts 11-6
On-Event Interrupts 11-7
ON ERROR GOTO Statement 11-8
OFF ERROR Statement 11-9
ON CTRL/C GOTO Statement 11-10
OFF CTRL/C Statement 11-11
ON #n GOTO Statement 11-12
OFF #n Statement 11-13
ON KEY GOTO Statement 11-14
OFF KEY Statement 11-15
ON PORT Statement 11-15
OFF PORT Statement 11-15
ON PPORT Statement 11-15
OFF PPORT Statement 11-15
ON SRQ GOTO Statement 11-16
OFF SRQ Statement 11-17
ON PPOL GOTO Statement 11-18
OFF PPOL Statement 11-19
SET CLOCK Statement 11-19
ON CLOCK Statement 11-19
OFF CLOCK Statement 11-19
SET INTERVAL Statement 11-20
ON INTERVAL Statement 11-20
OFF INTERVAL Statement 11-20
RESUME Statement 11-21
WAIT FOR EVENT Interrupts 11-22
WAIT Statement 11-24
WAIT Time Statement 11-24
WAIT FOR KEY Statement 11-25

(continued on page viii)

WAIT FOR SRQ Statement 11-26
WAIT FOR PPOL Statement 11-27
Errors and Error Handling 11-28
Fatal Errors 11-28
Recoverable Errors 11-28
Warning Errors 11-29
Error Variables 11-29
Interrupt Control Statements 11-30
Interrupt Processing Program Examples 11-31

12 SUBROUTINES 12-1
Introduction 12-3
Overview 12-3
Using Internal Subroutines 12-4
Using External Subroutines 12-5
Software Requirements 12-6
Subroutine Names 12-6
Assembly Language Subroutines 12-7
Assembly Language Error Handler 12-7
FORTRAN Subroutines 12-8
Subroutine Format 12-9
Introduction 12-9
Parameter Passing Mechanism 12-9
The Parameter Decoding Subroutine, F$RGMY 12-10
Standard Assembly Language Subroutine Linkage Mechanism 12-10
Multiple Subroutine Entry Points 12-13
Subroutine Parameter Formats 12-14
Introduction 12-14
Basic Internal Data Formats 12-14
Relationship Between Parameter Syntax And Parameter Format 12-17
Passing String Values To and From Subroutines 12-19
Conversion of Strings to Internal Form 12-19
Conversion of Strings from Internal Form. 12-20

13 PROGRAM CHAINING 13-1
Introduction 13-3
Overview 13-3
Statement Definitions 13-4
RUN Program Statement 13-4
RUN WITH Statement 13-6
EXEC Statement 13-6

(continued on page ix)

COM Statement 13-8
Virtual Arrays in Chained Programs 13-9
Introduction to Virtual Array Chaining 13-9
Example of Chaining a Virtual Array 13-10
Using Sequential Data Files in Chained Programs 13-12

14 TOUCH SENSITIVE DISPLAY 14-1
Description 14-3
Introduction 14-3
Using the Display for Output 14-4
The ASCII Character Set 14-5
Alternate Character Sets 14-6
Display Output Statements 14-7
PRINT Statement 14-7
PRINT USING Statement 14-8
CHR$() String Function 14-9
TAB String Function 14-9
CPOS String Function 14-10
Special Display Control Characters 14-12
Display Control Sequences 14-13
ANSI Compatible Display Control Sequences 14-14
Display Control Character Sequences 14-16
Cursor Positioning and Display Scrolling 14-17
Cursor Movement 14-17
Display Scrolling 14-18
Erasing 14-18
Mode Commands 14-19
Character Visual Attributes 14-20
Field Attributes 14-22
Character Attributes 14-25
Non-destructive Display Character 14-26
Character Size 14-27
Character Graphics 14-28
Keyboard Disable and Enable 14-32
Using the Display for Input 14-33
Display Input Statements 14-34
KEY Variable 14-34
ON KEY and OFF KEY Statements 14-35
WAIT FOR KEY Statement 14-36
An Interactive Display Program 14-39

(continued on page x)

15 PROGRAM DEBUGGING 15-1
Introduction 15-3
Overview 15-3
Debugging Tools 15-4
TRACE ON Statement 15-4
Line Number Tracing 15-5
Variable Tracing 15-8
Other Trace Options 15-11
TRACE OFF Statement 15-13
STOP ON Statement 15-13
STEP Command 15-14
CONT TO Command 15-15

16 ERROR MESSAGES 16-1

APPENDICES
A Internal Structure of Variables A-1
B IEEE 488 Bus Messages B-1
C Wbyte Decimal Equivalents C-1
D Parallel Poll Enable Codes D-1
E Display Controls E-1
F Graphics Mode Characters F-1
G ASCII & IEEE Bus Codes G-1
H Assembly Language Error Handler H-1
I Fortran Interface Runtime Library I-1
J Virtual Array Dimensioning Program J-1
K Graphics K-1
L Supplementary Syntax Diagrams L-1
M Glossary M-1
N Reserved Words N-1

INDEX

Section 1
How To Use This Manual

CONTENTS

Introduction 1-3
BASIC Manual 1-5
BASIC Reference Manual 1-6
How to Read Syntax Diagrams 1-7
Notation Conventions 1-9

INTRODUCTION

When originally developed at Dartmouth College in 1964 the BASIC computer language was intended to be a starting point for students learning the rudiments of computer programming. The name BASIC is an acronym for "Beginners All Purpose Symbolic Instruction Code". BASIC was intended to be simple and easy to learn.

Today, the BASIC Language is much more than a beginners language. BASIC has evolved into a powerful, but easy-to-use programming language that is used for more general-purpose programming than any other computer language. BASIC is the de-facto standard for microcomputers and instrument controllers.

This manual documents the BASIC Language option for the Fluke 17XXA series Instrument Controllers. The BASIC Language defined for the 17XXA series is a superset of Standard (ANSI) BASIC with the following enhancements:

- IEEE-488 bus operation and management.
- Graphics capability for the 17XXA series Instrument Controllers.
- Virtual Array storage (random-access disk files).
- BASIC program linkage to Assembly Language or FORTRAN subroutines.
- Operator feedback from the Touch-Sensitive Overlay (TSO).
- System Variables for time-of-day, date, time, error handling, available memory, touch-sense key, and file length.
- Direct port input and output.
- Device and file management functions.
- Error handling.

If you have never written a BASIC program before, you should obtain and read the following reference:

BASIC from the Ground Up
David E. Simon
©1978 Hayden Book Company, Inc
Rochelle Park, NJ
Catalog #: 5760-1
ISBN 0-8104-5117-4

After familiarizing yourself with Standard BASIC, read the tutorial sections of this manual.

If you are familiar with Standard BASIC, the various manual sections may be consulted individually as required.

BASIC MANUAL

This manual is divided into these major sections:

Section 1 How To Use This Manual
Describes the organization of the various manual sections, syntax diagrams and notation conventions.

Section 2 BASIC Conventions
This section describes the differences between Fluke BASIC and Standard BASIC, running and exiting the BASIC Interpreter program, BASIC operating modes and file names.

Section 3 System Functions
Describes how to perform FDOS functions from within the BASIC environment.

Section 4 Writing And Editing A Program
Describes the program writing process and the uses and usage of the BASIC Editor and the System Editor.

Section 5 Storing And Running A Program
Describes how to store, load, and run a stored program.

Section 6 Data, Operators, And Expressions
Describes the various data types, variables, operators and expressions used in Fluke BASIC.

Section 7 Data Storage
Describes data storage techniques using main memory and virtual arrays as well as the advantages and disadvantages of both.

Section 8 Functions
Describes and explains the various string and mathematical functions available in Fluke BASIC

Section 9 IEEE-488 Bus I/O
Describes and explains the use of the IEEE-488 Bus Control statements.

Section 10 RS-232 Serial I/O
Describes the RS-232 serial I/O ports and how to use them from a BASIC Language program.

Section 11 Interrupt Processing
Describes using the interrupts as a means of control and error-recovery.

Section 12 Subroutines
Describes the use of subroutines in a BASIC program. How to use subroutines written in Assembly Language and FORTRAN with programs written in BASIC. How to pass parameters to and from each program.

Section 13 Program Chaining
Describes how to run BASIC programs, and how to chain (connect) to other BASIC, Assembly Language, or Command files.

Section 14 Touch Sensitive Display
Describes the display control codes, formatting, and graphics. Shows how to use the Touch Sensitive Display for operator interface.

Section 15 Program Debugging
Describes techniques and tools used to correct errors in BASIC programs.

Section 16 Error Messages
A numerical listing of error codes and their meanings.

APPENDICES
A collection of reference material and a glossary of terms.

INDEX
A thoroughly cross-referenced listing of items and topics explained in this manual.

BASIC REFERENCE MANUAL

The BASIC Reference Manual contains an alphabetical listing of all commands, statements, functions, and system variables. If you are familiar with standard BASIC, you will be comfortable working with this manual most of the time.

HOW TO READ SYNTAX DIAGRAMS

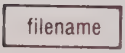
A syntax diagram is a graphical representation of how to construct a valid command or statement in a programming language. It is a kind of "shorthand" way of writing down all the rules for using the elements of a language. Since they are used throughout this manual, learning how to read them can be a great time saver.

Words inside ovals must be entered exactly as they are shown.

Words inside boxes with rounded corners indicate a single key must be pressed, such as RETURN or ESC.

This indicates a space in the statement. (Press the spacebar.)

To create a control character, hold down the control key (CTRL), then press the other key. This one is a Control C; it causes a break in the program.

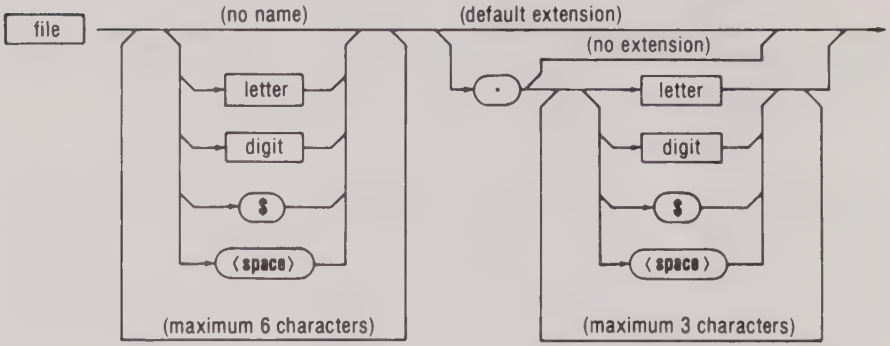


A box with lower-case words inside means that you supply some information. In this case, you would enter a filename.

(explanation)

Words in parentheses are explanations of some kind. They give added information about the nearest block or path.

From the left, any path that goes in the direction of the arrows is a legitimate sequence for the parts of a statement. This sample shows the correct syntax for naming a file. The translation is given below.



A line exits the top of this diagram with no keyboard input. This indicates that it is possible to not specify the filename or its extension. In this case, the file would have no name, and the system would assign a default extension.

Further down the diagram, you can see that there are other possibilities. You can choose up to 6 characters for the filename, moving once through the loop for each character chosen. Up to three characters can be chosen for the extension.

Filename can be any combination of letters, digits, the $ sign, and spaces.

The filename and extension must be separated by a period, as shown in the oval block at the top center.

The remark "no extension" means that it is not necessary to specify an extension, even though a file name is given. Notice however, that this remark occurs after the period, so the period is necessary if a name is specified.

Here are some examples of valid filenames according to the syntax illustrated in the diagram:

TESTIN.$3A 1722A.RAC $$$$$$.$$$

NOTATION CONVENTIONS

The conventions listed here are used for illustrating keyboard entries and to differentiate them from surrounding text. The braces, { } ; brackets, [ ]; and angle brackets, ⟨ ⟩ ; are not part of the keystroke sequence, but are used to separate parts of the sequence. Do not type these symbols.

⟨xxx⟩ Means "press the xxx key".
Example: ⟨RETURN⟩ indicates the RETURN key.

⟨xxx⟩/y Means "hold down key xxx and then press y".
Example: ⟨CTRL⟩/C means to hold down the key labeled CTRL and then press the key labeled C.

[xxx] Indicates an optional input.
Example: [input filename] means to type the name of the input filename if desired. If not, no entry is required, and a default name will be used.

xxx Means to type the name of the input as shown.
Example: BASIC means to type the program name BASIC as shown.

{xxx} Indicates a required user-defined input.
Example: {device} means to type a device name of your choice, as in MF0: for floppy disk drive 0.

(xxx) This construction has two uses:

1. As a separate word, (xxx) means that xxx is printed by the program. Example: (date) means that the program prints today's date at this point.

2. Attached to a procedure or function name, (xxx) means that xxx is a required input of your choice; the parentheses are a required part of the input. Example: TIME(parameter) means that a procedure specification is the literal name TIME followed by a parameter that must be enclosed in parentheses.

Section 2
BASIC Conventions

CONTENTS

Introduction 2-3
Overview 2-3
Exceptions to ANSI Standard BASIC 2-4
Running the BASIC Interpreter Program 2-5
Exiting BASIC 2-6
BASIC Operating Modes 2-7
  Immediate Mode 2-7
  Edit Mode 2-9
  Run Mode 2-9
  Step Mode 2-10
File Names 2-11

INTRODUCTION

The English language has certain conventions in syntax, grammar, and conjugation (to name a few). English also has certain exceptions to these rules. BASIC, a computer language but a language nonetheless, has its own set of rules and exceptions.

OVERVIEW

This section of the manual describes the differences between Fluke BASIC and ANSI Standard BASIC, a few things to watch out for when converting a program written in Standard BASIC to Fluke BASIC, how to run the BASIC Interpreter Program, how to exit the BASIC Interpreter Program and an explanation of the BASIC Interpreter Program's various operating modes.

The BASIC Language Interpreter program for the 17XXA Series Instrument Controller is based upon ANSI Standard X3J2/77-26 Minimal Basic. (Within this manual, the 17XXA Series Instrument Controller will be referred to as the Instrument Controller or simply, the Controller.) A comprehensive set of additional command statements for control of IEEE-488 bus instrumentation has been added as well as string-data operation, mass-storage control commands, debugging features, FDOS operations, and other extensions. Also included are statements for chaining multiple programs in sequence and passing variables between programs. In addition to these functions and features, the BASIC interpreter allows use of Texas Instruments TMS-99000 Assembly language subroutines and FORTRAN subroutines.

EXCEPTIONS TO ANSI STANDARD BASIC

Fluke BASIC is an enhancement of ANSI Standard Minimal BASIC. The enhancements include many statements for instrument control. An ANSI Minimal BASIC program will run on the Instrument Controller unless there is conflict in one of the following areas:

- Dimensioned arrays must be defined by a dimension statement prior to use in a program.
- LOG is the logarithm to base 10. LN is an additional BASIC function for natural logarithms (base e).
- A GOTO within a FOR-NEXT loop is not allowed if the GOTO transfers control permanently outside the loop.
- The print format used for floating-point numbers greater than 7 digits in length is .1 to 1, instead of 1 to 10 as specified in the ANSI standard. For example, .01234567 is printed as .1234567E-01, instead of 1.234567E-02.
- TAB(X) will position the cursor at X+1 instead of X.
- Variables are assigned in sequence, upon validation, in a multiple-variable input list. The ANSI standard allows variable assignment only upon validation of the entire input list.
- The optional base of 1 for a dimensioned array is not supported. Fluke BASIC arrays have a base of 0. For example, A(10) has eleven elements numbered 0 through 10.
- User defined functions must have arguments. For example, DEF FNA(X) = expression is allowed while DEF FNA = expression is not.
- Multiple commands are allowed on a single line when separated by the \ character.
- Spaces may not appear between GO and SUB of GOSUB or between GO and TO in GOTO.
- IEEE-488-1980 Instrumentation Bus control statements.

RUNNING THE BASIC INTERPRETER PROGRAM

BASIC is run from the FDOS prompt by typing the BASIC Interpreter program file name: BASIC. Refer to the 1722A System Guide for information on how to run the FDOS program. When BASIC is run, the program produces the following display.

```
BASIC Version (x.y)
Ready
```

NOTE

Verify that the version number given for "x.y" agrees with the version number at the beginning of this manual. If not, contact a Fluke Customer Service Center for advice.

In addition, BASIC may be run at Power-On or when RESTART is pressed. This is done with an active Command File that contains the file name BASIC. Refer to the 1722A System Guide for information on the Command File, STRTUP.CMD.

EXITING BASIC

There are four ways to exit BASIC. Each returns control to FDOS, resulting in the FDOS〉 prompt.

- From Immediate Mode (READY prompt) type EXIT and press RETURN. This allows input/output operations to finish, closes files, deletes the BASIC program in memory, and returns to the FDOS Command Line Interpreter. If the BASIC program in memory has been edited and not SAVE'd or RESAVE'd, the following prompt is issued:

  Program has been changed but not saved. Really EXIT?

- If answered yes, the BASIC Interpreter will delete the program in memory and exit to FDOS. If answered no, the EXIT routine is aborted.

- Execute an EXIT statement within a running program. This allows input/output operations to finish, closes files, deletes the BASIC program in memory, and returns to FDOS.

- Press RESTART. This transfers control to the self-test and load routine, terminates all operations of the Instrument Controller, loads FDOS, and processes the Command File (STRTUP.CMD), if there is one. The Instrument Controller User Manual contains information on the Command File.

CAUTION

When RESTART is pressed, any I/O in progress is aborted and the file directory may be destroyed.

Enter 〈CTRL〉/P (press and hold the CTRL key, press the P key then release both keys). This closes all files immediately, deletes the BASIC program in memory, and loads FDOS. Output operations are not finished before the files are closed.

CAUTION

If a BASIC program is in memory and control is returned to FDOS, that program will be deleted from memory.

BASIC OPERATING MODES

BASIC has four modes: Immediate, Edit, Run, and Single-Step. The "Ready" prompt is displayed whenever BASIC is in Immediate Mode and ready to accept keyboard input. This is the mode that BASIC starts in whenever it is entered from FDOS. Other modes are accessible only from Immediate Mode.

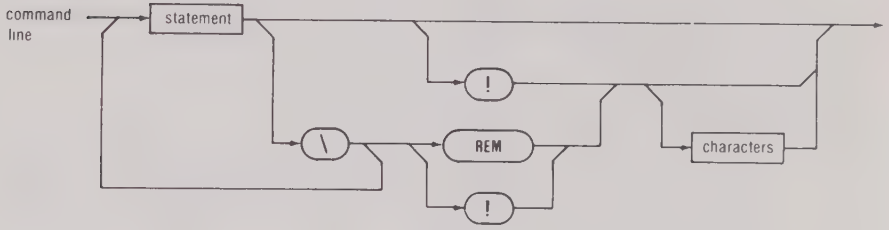
Immediate Mode

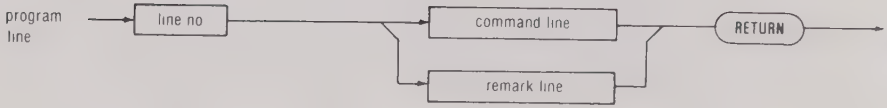
An entry to BASIC is always into Immediate Mode, identified by the prompt "Ready". Immediate mode may be accessed from any other BASIC mode by typing ⟨CTRL⟩/C. Also, a running program will return to Immediate Mode whenever the program executes a STOP or END statement or if a fatal error occurs.

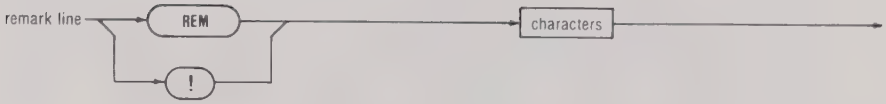
Exit from Immediate Mode to FDOS is accomplished by typing EXIT, or by typing ⟨CTRL⟩/P. A "Really Exit?" prompt is given if a program exists in main memory that has been edited but not SAVE'd or RESAVE'd. If the prompt is answered with "yes", any program that was in main memory is erased by the exit to FDOS, otherwise Immeditate Mode is re-entered.

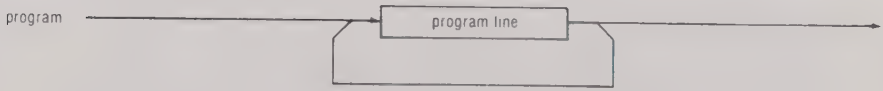
In Immediate Mode, command lines or program lines may be typed. A command line consists of one or more BASIC statements; these statements are typed either without a line number or with a line number of zero. Multiple BASIC statements are separated by a backslash character (\). An Immediate Mode command is executed immediately upon pressing RETURN. Some BASIC commands can be executed only in Immediate Mode. These commands or statements are identified as Immediate Mode Commands or IMCs. Subsequent sections cover them in detail.

A program line is a command line preceded by a line number between 1 and 32767. A program is a meaningful sequence of program lines. The following diagrams illustrate these points:









Edit Mode

The entry to Edit Mode is made from Immediate Mode by entering EDIT, or EDIT followed by a valid line number. Editing begins with the lowest numbered line of the program memory unless a line number is specified. The exit from Edit Mode to Immediate Mode is accomplished by typing ⟨CTRL⟩/C.

The editor provided as a part of Fluke BASIC is an easy-to-use screen-oriented editor. Edit Mode allows the user to insert, delete, or modify the characters that make up program lines in main memory. The editing (arrow) keys in the upper right corner of the keyboard plus the BACK SPACE, ⟨CTRL⟩/U, Carriage Return, and LINE FEED keys control the movement of the cursor and the deletion of text. The remaining keys are used for text entry. Edit Mode stores program lines in main memory for subsequent use by other modes.

Run Mode

The transition to Run Mode is made from Immediate Mode by typing RUN, or RUN "file name". Refer to the Input and Output section for a discussion of file name prefixes that may be used to identify the location of the program file to be run. If a file name is specified, the file is first located on the specified or default file storage device and it is then loaded into main memory.

When a program is present in main memory and BASIC is in Immediate Mode, Run Mode may also be entered by an Immediate Mode branch command: GOTO, GOSUB, ON GOTO, or ON GOSUB followed by a valid line number. In addition, when a running program has been stopped by ⟨CTRL⟩/C, STOP, STOP ON, or CONT TO, Run Mode may be resumed by typing CONT or CONT TO followed by a valid line number. These concepts are discussed in the section Program Debugging.

BASIC exits RUN Mode and returns to Immediate Mode under any of the following conditions:

- The program executes a STOP or END statement.
- ⟨CTRL⟩/C is entered at the programmer keyboard.
- The front panel ABORT button is pressed.
- A fatal error occurs.

In Run Mode, program lines are automatically executed in the sequence defined by line numbers, except as directed by branch instructions in statements.

Step Mode

Step Mode is entered from Immediate Mode by typing the STEP command after a breakpointed stop set by CONT TO or STOP ON. A RUN, CONT, CONT TO, GOTO, GOSUB, ON GOTO, or ON GOSUB command will cause BASIC to enter Run Mode; typing any other command will cause BASIC to enter Immediate Mode.

In Step Mode, one program line is executed after each RETURN key entry, followed by a "STOP AT LINE line number" message. Step Mode is discussed further in the Program Debugging section.

FILE NAMES

Programs and data files on the Instrument Controller are stored on the floppy, electronic disk or other file-structured media by name. File names consist of 1 to 6 letters, numbers, spaces, or $ characters. File names may be extended by a period character, followed by up to 3 letters, numbers, spaces, or $ characters. Most usage of file names in program statements requires that they be enclosed in single or double quotes. Some examples of file names are:

```
TEST
DATA.473
$25795.98
TESTB2.H44
```

Some file name extensions have special meaning to the Instrument Controller:

.BAL	Lexical form BASIC programs
.BAS	ASCII-text form BASIC programs
.FD2	Binary utility programs
.HLP	System data files (Help)
.SYS	System binary programs
.CMD	Command file

Following are points to remember about file names for programs:

- A program is normally saved in a file with a ".BAS" extension when an extension is not specified and the SAVE command is used.
- A program stored via the SAVEL command is saved with a ".BAL" extension. See the SAVEL statement.
- The system will look for a file with the extension ".BAL" when seeking a program with a file name that has no specified extension.
- The system will look for a file with the extension ".BAS" when seeking a program with a file name that has no specified extension for which a ".BAL" version could not be found.
- The location of the file may be specified as a prefix to the file name. Use "MF0:" for the floppy disk, or "ED0:" for the optional electronic disk. For example: SAVE "MF0:TEST.T12".
- The default file location when not specified is called the System Device. The Input and Output section includes a discussion of this concept.

Section 3
System Functions

CONTENTS

Introduction 3-3
Overview 3-3
System Functions for File Management 3-4
  COPY Statement 3-5
  DIR Statement 3-5
  EDIR Statement 3-5
  KILL Statement 3-6
  QDIR Statement 3-6
  RENAME Statement 3-6
Device Management Functions 3-7
  ASSIGN Statement 3-7
  PACK Statement 3-7
  PROTECT Statement 3-7
  UNPROTECT Statement 3-8

INTRODUCTION

File and device management are two important functions of a disk operating system. The File Utility Program (FUP) is a general-purpose file and device manager program accessible from FDOS. FUP commands provide functions such as: device assignment, file copy, file deletion, file rename, and media formatting.

OVERVIEW

The BASIC Interpreter program has a family of file and device management statements. These statements may be used in the immediate mode or as part of a program. The System Function statements do not provide all of the capabilities of FUP nor are they intended to replace FUP. Instead they are intended to allow a running program to manage its own files without the necessity of leaving the BASIC environment. FUP is described in Section 4 of the System Guide.

SYSTEM FUNCTIONS FOR FILE MANAGEMENT

During the execution of a BASIC program, it may be necessary or desirable to allow the BASIC program to manipulate (manage) some or all of the files on the various mass storage devices. For example, a data collection program that operates through the course of a day may use a copy of the master data file for each new run of the data collection program. Once the data collection portion program has been successfully completed, the filename extension of the old master data file is changed to ".bak" using the rename command, then the copy is renamed to be the master data file. The following example makes this concept clearer.

Starting conditions:

File: "mydata.vrt" the master file

Run program

Copy "mydata.vrt" to nudata.vrt.

Collect data, store on "nudata.vrt"

If data collection fails, the original data file has been preserved.

Data Collection done

```
rename "mydata.vrt" as "mydata.bak"
rename "nudata.vrt" as "mydata.vrt"
```

Files are ready for the next run.

As you can see, the original data file is always protected from being accidentally destroyed by always using a copy of the orginal file, then renaming it after a successful program run.

The File Management functions are presented here in alphabetical order. Each of the functions is summarized in this section. For a more complete treatment, along with a syntax diagram, refer to the Reference volume of this manual set.

NOTE

Many of the examples to follow have explicitly mentioned file names. Since these file names are strings, the BASIC interpreter program requires that they be enclosed in quotation marks. Any of these quoted strings may also be arbitrary string-valued expressions.

COPY Statement

The COPY statement copies one or more files to the screen or to another file. The general form of the COPY statement is:

```
COPY "old.ext" TO "new.ext"   ! copies to another file

  or

COPY "foo.ext'   ! copies to the screen
```

Remember that both filenames are strings, therefore both must be enclosed in double quotes. If no device is specified for "thisfile.ext" or "thatfile.ext" then the system device (SY0:) is assumed.

DIR Statement

The DIR statement prints the directory of a file-structured device such as MF0: or ED0:. The general form of the DIR statement is:

```
DIR "mf0:"   ! lists on the screen

  or

DIR "mf0:" TO "dirfil.txt"   ! copies directory to file
```

If the device name is omitted, the directory of the system device SY0: is used. The format of the statement output resembles the /L directory listing in FUP.

EDIR Statement

The EDIR statement is similar to the DIR statement except that unused and tentative directory entries are also listed. The statement syntax is identical to the DIR statement. The display format resembles the /E listing in FUP.

KILL Statement

The KILL statement deletes a specified file. The KILL statement takes the following form:

```
KILL "mf0:file.bas"
```

If a device is not specified, the system device SY0: is used.

QDIR Statement

The QDIR statement prints a short or quick directory listing of only the filenames. File size, protection codes or creation dates are not given. The statement syntax is identical to the DIR statement. The display format lists the filenames six per line.

RENAME Statement

The RENAME statement allows a file to be given a new name. The general form is:

```
RENAME "oldfil.ext" AS "nufile.ext"
```

where oldfil.ext is the current filename and nufile.ext is the new filename to be used. Again, SY0: is assumed if no device specification is given with either filename.

DEVICE MANAGEMENT FUNCTIONS

The device management functions permit a BASIC program to perform additional FUP-like functions such as assigning a device, packing to remove unused blocks, and setting or un-setting write protection for a file.

ASSIGN Statement

The ASSIGN statement permits the user or a running program to assign a particular device as the system device, SY0:. The general form of the statement is:

```
ASSIGN "mf0:"   ! assign the floppy disk as SY0:

  or

ASSIGN "ed0:"   ! assign the E-disk as SY0:
```

An error will be reported if the named device is not a known file-structured device.

PACK Statement

The PACK statement is used to remove unused sectors from the specified mass storage device. The general form of the PACK statement is:

```
PACK "mf0:"
```

If no device is given, the system device, SY0:, is used. If the PACK statement is used in the immediate mode, a warning prompt is issued to alert the user that a lengthy operation is in progress.

PROTECT Statement

FDOS permits files to be selectively write or delete protected. This means that protected files may not be deleted or over-written (e.g. via the KILL or OPEN AS NEW FILE statements). The general form of the PROTECT statement is:

```
PROTECT "myfile.ext"
```

which sets the write protection for the named file.

UNPROTECT Statement

The UNPROTECT statement removes the protection set by the PROTECT statement. The UNPROTECT statement is the complement of the PROTECT statement. The general form of the UNPROTECT statement is:

```
UNPROTECT "myfile.ext"
```

which un-sets the write protection for "myfile.ext".

Section 4
Writing and Editing a Program

CONTENTS

Introduction 4-3
Overview 4-3
Special Function Keys 4-3
  Delete Key 4-3
  PAGE MODE Switch and NEXT PAGE Keys 4-3
  ⟨CTRL⟩/Key-Modifier 4-4
Writing a Program 4-6
Entering a Program 4-7
  Entering a Program from the Immediate Mode 4-7
    LIST Command 4-9
    DELETE Command 4-10
    REN Command 4-11
    Changing the Sequence of Program Lines 4-14
  Entering a Program From the Edit Mode 4-15
  Entering a Program Using the System Editor program (EDIT.FD2) 4-15
Editing a Program 4-16
  The BASIC Editor 4-17
    Edit Mode Keys 4-19
    Additional Editor Features 4-22

INTRODUCTION

Writing a program is the first step toward getting the controller to do your bidding. This section describes the process of entering a program into the memory of the controller, and making corrections to that program.

OVERVIEW

A BASIC program is any meaningful sequence of Fluke BASIC statements that (in Run Mode) directs the Instrument Controller and its associated instrumentation to accomplish a desired task. The program statements that make up a BASIC program are made up of BASIC language elements and their optional modifiers or arguments.

SPECIAL FUNCTION KEYS

The Instrument Controller's keyboard has several function keys which extend the capability of the keyboard beyond the simple task of entering text. These keys are described in the following paragraphs.

DELETE Key

The DELETE key deletes the character immediately to the left of the cursor. If held down, it will repeatedly backspace and delete characters until all characters to the left of the cursor are deleted.

PAGE MODE Switch and the NEXT PAGE Keys

PAGE MODE is an alternate action switch that selects and cancels Page Mode. The Page Mode indicator on the switch is ON when Page Mode is selected. When PAGE MODE is enabled, the scroll feature is disabled and display is limited to one full screen. The NEXT PAGE switch displays the next full screen from the top line down.

⟨CTRL⟩/Key-Modifier

The ⟨CTRL⟩/ key is a key-modifier that is always used in conjunction with another key, much like the shift key. Several of the ⟨CTRL⟩/+ key combinations have special meanings to BASIC editor. They are:

- ⟨CTRL⟩/C returns BASIC to Immediate Mode resulting in a "Ready" display prompt. In addition, a common use of ⟨CTRL⟩/C is to terminate a program test run during program development. Some further considerations are:

- If an IEEE-488 Bus operation is in progress, ⟨CTRL⟩/C stops the transfer immediately.

- In Edit Mode, ⟨CTRL⟩/C will store the current line as edited if it is syntactically correct. If not, the line is stored as it was before editing.

- The ABORT button is treated by BASIC as a ⟨CTRL⟩/C; in addition, a Device Clear message is sent to the IEEE-488 instrument ports.

- In Run Mode, execution of an ON CTRL/C GOTO statement inhibits a return to Immediate Mode. Instead, control is transferred to the program line referenced. See the Interrupt Processing section.

- ⟨CTRL⟩/ P stops any input/output operation in progress and transfers control back to FDOS. This action is taken regardless of what the Instrument Controller was doing previously.

CAUTION

⟨CTRL⟩/P deletes the user program in memory. Also, ⟨CTRL⟩/P might not properly terminate a file transfer operation in progress (the buffers are not "flushed").

- ⟨CTRL⟩/R restores the last line typed in the immediate mode.

- This line may then be edited using the Edit Mode keys editing features of the command line interpreter.

- The edited line may be executed by pressing ⟨RETURN⟩, regardless of the cursor position.

- Each time ⟨CTRL⟩/R is pressed, the last line entered is displayed, until the first line entered has been reached.

- Likewise, each time ⟨CTRL⟩/F is pressed, the Controller steps the command line memory forward and displays the next line, relative to the current line.

- Additional editing commands are described in Section 6 of the System Guide.

- ⟨CTRL⟩/S stops the display from scrolling. ⟨CTRL⟩/Q allows the display to continue scrolling. If a running program displays enough data to make the display scroll, ⟨CTRL⟩/S will stop the display for study. The program will suspend output to the display until ⟨CTRL⟩/Q is entered.

- There are some differences between using ⟨CTRL⟩/S and ⟨CTRL⟩/Q, and using Page Mode with the NEXT PAGE key for control of display scrolling:

- NEXT PAGE scrolls the next page onto the screen.

- After a ⟨CTRL⟩/S has been pressed, there is no indication of the status of the system; e.g., no indicator as on the PAGE MODE key. Thus, ⟨CTRL⟩/S can cause the keyboard to appear "dead". ⟨CTRL⟩/Q will clear this condition.

- ⟨CTRL⟩/T erases the display, moves the cursor to the upper left corner, enables single-size characters, disables graphics mode and all other display enhancements, and enables the keyboard. It then sends a Carriage Return and Line Feed. The section on The Touch Sensitive Display describes display enhancements.

- ⟨CTRL⟩/ U deletes the current line.

WRITING A PROGRAM

There are many techniques used in writing a computer program. Everyone has their own pet method. One common method is to simply sit down in front of the keyboard and start writing. While this method is probably the most direct, it can produce a program that, in retrospect, is difficult to maintain and/or troubleshoot.

In contrast, a program that is well thought out and documented during the writing process is generally easier to troubleshoot and maintain. The following points highlight this process.

1. Make a general statement of the task. It should be as general as possible.

2. Break this general statement into subtasks and sub-sub tasks. Repeat this until each subtask represents one activity.

3. Describe each subtask using English-like or BASIC words (pseudocode) to describe what needs to be done. This is the time to think out program flow and execution sequence.

4. Write the actual program in BASIC from the pseudocode produced in the previous step. Use the REM statement to describe the activity taking place within each program block (subtask).

5. Debug the program blocks produced above. If possible, debug each one separately from the rest. It is much easier to isolate errors in this fashion.

6. Connect the previously debugged program blocks into the final program, write any connective subroutines or program lines, and perform the final debugging.

7. The program shoud be ready to run at this point.

ENTERING A PROGRAM

Entering a program is the process of typing a program into the controller's memory. There are three ways of doing this.

1. From the Immediate Mode, enter the program lines by typing them on the keyboard.

2. Use the Edit Mode provided by the BASIC Interpreter program. This is the easiest way to enter a program into the Controller's memory.

3. Use an external editor program to create an ASCII text file that the BASIC Interpreter program can then load and run.

Entering a Program from the Immediate Mode

From the Immediate Mode, type the line number, followed by a space and the rest of the program line.

- Program lines may be typed (entered) in any order. Regardless of the sequence in which program lines are entered, BASIC will store and execute them in line number order.

- Program lines may be typed in upper- or lower-case. BASIC converts lowercase entries for program statement keywords and variables to uppercase (capitals).

- Each program line must begin with a line number.

- Each program line must not exceed 79 printable characters in length, including the line number, plus one character space for the carriage return, for a total of 80 characters.

- If you accidentally (or on purpose) use the same line number twice, the last line entered is the line that is kept.

- Some checking is done for syntax errors (like mismatched parentheses).

- If you make an error while typing, use the delete key to make corrections, or use the Edit mode of the Basic Interpreter program. You can also retype the entire line.

- To insert a line, use any line number between the two lines where it needs to be placed. BASIC will store the new line in the proper sequence. This is why BASIC line numbers are normally incremented by 10.

- To replace a line, type the new line with the line number of the old line.

- To delete a line, type the line number only. Then press ⟨RETURN⟩. See also DELETE below.

- Use the LIST command to observe the program as it currently exists in memory.

- Each line may contain remarks only, or it may contain one or more statements optionally followed by remarks.

- When all program lines have been entered, type ⟨RUN⟩ to run the program.

- Program lines may contain multiple statements (commands) which are separated by the backslash character (\ ) but must not exceed 79 printable characters plus the carriage return.

- In general, BASIC executes all the statements on a multiple statement line before going on to the next line. The IF-THEN statement discussed later is one important exception to this general rule.

This discussion presents Immediate Mode commands which aid in creating or modifying a program. They are: LIST, DELETE, and REN.

LIST Command

The LIST command displays a program or a portion of a program in line number order.

- Display starts at the first line of a program and proceeds to the last line if line numbers are not specified.
- One line is displayed if a single line number is specified. The command is ignored if the line does not exist.
- A portion of a program is displayed if two line numbers are specified. The display will be the lines with numbers between and including the specified lines numbers if they exist.
- If the portion to be listed is larger than one display page (16 lines), the display will scroll upwards until the last line specified has been displayed.
- Use Page Mode and the NEXT PAGE key, or ⟨CTRL⟩/S and ⟨CTRL⟩/Q to stop and restart the display. These functions are discussed earlier in this section.

The following examples illustrate common uses of the LIST command:

COMMAND	RESULTS
`LIST`	`Displays the entire program in memory from the first line.`
`LIST 500`	`Displays only line 500 of the program, if it exists.`
`LIST 600-800`	`Displays a program segment beginning with 600 and 800, inclusive.`

DELETE Command

The DELETE command deletes part or all of a program from memory.

- The entire program is deleted when ALL is specified.
- DELETE ALL also deletes Common Variables (see the COM statement in the Program Chaining section of this manual and in the Reference volume of this manual set).
- One line is deleted if a single line number is specified. The command is ignored if the line does not exist.
- One line may also be deleted by typing the line number only, followed by pressing RETURN.
- The portion of the program between and including specified lines is deleted if two line numbers are specified.

The following examples illustrate common uses of the DELETE command.

COMMAND	RESULTS
`DELETE`	`No action`
`DELETE ALL`	`Deletes entire program`
`DELETE 100`	`Deletes only line 100`
`DELETE 200-300`	`Deletes lines 200 through 300`
`400 <RETURN>`	`Deletes line 400`

REN Command

The REN command changes the line numbers of some or all of the program lines in memory. Renumbering is useful to make room for additional program lines.

- REN cannot change the order of program lines.
- REN changes all references to line numbers (i.e., GOTO, GOSUB, etc.) in the program to reflect the new line numbers.
- All items shown in the syntax diagram are optional except the command word REN.
- The entire program is renumbered when no line numbers are specified.
- One line is renumbered when a single line number is specified. The command is ignored if the line does not exist.
- A portion of the program is renumbered when two line numbers are specified.
- The line number following AS specifies the new starting number of the segment being renumbered. If this would rearrange the sequence of the program, a fatal error occurs and the line numbering remains unchanged.
- When AS is not specified, and a line number or range of line numbers is not specified following REN, the new starting line number is 10.
- When AS is not specified and a line number or range of line numbers is specified following REN, the new starting line number is the same as the old first line number of the range specified.
- The value of the integer expression following STEP must be positive. It defines the difference between any two consecutive, renumbered lines.
- If there is no STEP keyword, the line increment is 10.
- If the value of the integer expression following STEP is so large that the new line numbers would force the program to be rearranged, a fatal error occurs and the lines are not changed.

CAUTION

Renumbering from lower to higher line numbers (with more digits) may cause the renumbered lines to exceed the 79-character maximum line length allowed by BASIC. Restricting program lines to 74 characters maximum length will generally eliminate this problem. This exception is on long lines which include line number references (e.g., ON expression GOTO, IF-THEN-ELSE with line numbers, etc).

NOTE

Program lines containing the ERL (error line) function may have statements such as IF ERL = 200 THEN RESUME 400. The expression, the constant 200, is used as a line number reference. It is not changed during renumbering. It may need to be changed to the correct line number manually.

The following program is used in the renumbering examples below:

```
10   A = 1
20   PRINT A + A
30   A = A + 1
40   IF A <= 2 THEN 20
50   PRINT "Done!"
60   END
```

COMMAND	RESULT
`REN 60 AS 32767`	Change line 60 to read: `32767 END`

`REN 10-50 AS 5 STEP 5`	Change lines 10 through 50 to read:
	`5  A = 1`
	`10  PRINT A + A`
	`15  A = A + 1`
	`20  IF A <= 2 THEN 10`
	`25  PRINT "Done!"`
	Note the changed reference in line 20.

`REN`	This would in this case restore the program back to its original form.

`REN 60 AS 1000`	Renumber only line 60 as line 1000. An error results if any lines are numbered between 60 and 1001, since this would rearrange program sequence.
`REN 60 AS 1000 STEP 5`	Same as the previous example. STEP is ignored when only one line is renumbered.
`REN 10-500 AS 1000`	Renumber lines 10 through 500 to start at 1000, in steps of 10.

Changing the Sequence of Program Lines

REN cannot be used to change the sequence of program lines. The simplest way to change the sequence of program lines is to use Edit Mode (discussed below) and change line numbers individually. The process must also include changing line number references (GOTO, etc.) so they will refer to the correct lines. Edit Mode, however, is not best suited for the task when larger portions of a program are to be moved, because of the time required. As an alternate method:

1. Delete part of the program.
2. Renumber the remaining portion.
3. SAVE it under a temporary file name.
4. Retrieve the original program again, using OLD.
5. Repeat this procedure as needed.
6. Use the File Utility Program to merge these files together in the desired order. Refer to the Instrument Controller User Manual for details.

Here are some points to keep in mind while changing the order of program lines in this manner:

- FUP cannot merge lexical files. Use SAVE (or RESAVE but not SAVEL or RESAVEL) to save the program segments to be merged.
- A BASIC program cannot have more than one line for each line number. If two or more program segments which have internal line number conflicts are merged, the last occurrence of each line number will be all that remains when the merged file is processed by the BASIC interpreter. Ensure that all program segments use different blocks of line numbers, in the rearranged sequence that you need.
- Some of the line number references may be incorrect. Check and correct branching line number references before running the program.

Entering a Program From the Edit Mode

The BASIC Interpreter program has an easy-to-use screen-oriented editor which can be used for program entry or correction. To use the editor for program entry, type:

edit ⟨RETURN⟩

from the Immediate Mode (the program memory must be clear), then type the line number followed by the BASIC program line. Press ⟨RETURN⟩ when you reach the end of the line to return the cursor to the left edge of the screen and to open up a new line. The delete key deletes characters to the left of the cursor. Use it to make minor corrections to the program line.

While using the DELETE key is sufficient for simple corrections to a program line, this can become quite tedious as the changes necessary become more involved. Additional capabilities of the BASIC editor are described later in this section under the heading of The BASIC Editor.

Entering a Program Using the System Editor Program (EDIT.FD2)

The System Editor program can be used to enter a BASIC program and save it on a file-structured device such as a floppy disk. While this may seem like an extra step, the external editor's more powerful instruction set can make the task of entering a lengthy program easier.

Use the System Editor program by typing the following command from the FDOS prompt:

edit [filename]

The System Editor program has a complete set of editing commands including cursor movement, character insertion and deletion, string search, and search and replace. It is described in the Instrument Controller System Guide.

If you enter a program using the System Editor program and later edit it using the BASIC editor, be sure to use the SAVE or RESAVE (not SAVEL or RESAVEL) command to save the program on a mass-storage device (such as a floppy disk) before exiting BASIC. Doing this ensures that an ASCII image of the program is saved, which is readable by the System Editor program.

EDITING A PROGRAM

Editing is the process of selectively changing portions of a program. Usually editing is part of the debugging process when developing a program. BASIC, being an interpreted language, makes this process especially easy by allowing changes to be made and tested without leaving the BASIC environment. This is done using the EDIT Mode.

- Load the program into the controller's memory using the OLD or RUN commands, or by typing it in.
- Enter the EDIT Mode by typing EDIT from the READY prompt.
- Use the EDIT Mode commands to make the changes needed in the program.
- Once the changes have been made, return to the Immediate Mode by typing a CTRL/C.
- Type RUN to execute the program. This allows the effects of program modification to be observed immediately.
- If a problem persists, more powerful debugging techniques are discussed later in this manual.

If the changes required are lengthy, or involved, it may be easier to use the System Editor program to make the changes. This process is discussed briefly here and in more detail in the System Editor manual.

- Be sure to save an ASCII image of the program on the mass-storage device before exiting the BASIC interpreter program (use the SAVE or RESAVE command).
- Run the editor program by typing: edit programname.ext.
- The system editor cannot resolve line number conflicts, nor does it perform any syntax checking.
- When you are finished editing, be sure to save the modified file before exiting the editor program.

- Run the BASIC interpreter program by typing: BASIC from the FDOS prompt. Run your program by typing: RUN "programname" from the Ready prompt.

- If a problem persists, more powerful troubleshooting techniques are discussed later in this manual.

The BASIC Editor

The editor provided as a part of the Fluke BASIC interpreter program is an easy-to-use screen-oriented editor. The Edit Mode allows the user to create, delete, or modify the characters that make up program lines in main memory. Program lines are stored in main memory for subsequent use by other modes. The editing keys in the upper right corner of the keyboard plus the ⟨CTRL⟩/U, BACK SPACE, RETURN, and LINE FEED keys control the cursor and delete text. The remaining keys are used for text entry. This section describes the edit keys and their use along with other editor features.

- Edit Mode is entered from Immediate Mode by typing:

  EDIT
  or
  EDIT line number.

- Editing begins with the lowest numbered line of the program in memory unless another line is specified.

- No line number specification is used when beginning the edit of a new program when no other program is in memory.

- Program entry procedure is the same as in Immediate Mode.

- Immediate Mode commands and program statements cannot be executed while in Edit Mode.

- Exit from Edit Mode to Immediate mode by entering ⟨CTRL⟩/C.

- The special edit keys on the programmer keyboard are enabled.

- Up to 15 lines of the program in memory are displayed, beginning at the first line or at the line number given with the command.

- Edit Mode enables the user to scroll the cursor forward or backward in a program as well as right or left on program lines.
- Edit Mode enables the user to delete characters, portions of lines, or entire lines.
- Edit Mode also enables the user to duplicate entire program lines. The following examples illustrate the two different uses of the EDIT command.

COMMAND	RESULT
EDIT	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line. If no program exists, the display is cleared and the cursor is positioned to the upper left corner of the display.
EDIT 1000	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line greater than 999. If no program exists or the last line number is less than 1000, the display is cleared and the last line of the program is displayed at the top of the screen.

Edit Mode Keys

Some of the keys on the programmer keyboard have special functions that are enabled or modified in Edit Mode. Any key, if held down, performs its function repeatedly. Figure 4-1. describes the special functions of the Edit Mode Keys.

NOTE

Any edit command that will move the cursor from the current line or ⟨CTRL⟩/C is not accepted if the line does not pass a check for correct syntax. A blinking error message (e.g.: "Mismatched Quotes") will be displayed until the line is corrected.

Table 4-1. Edit Mode Keys

KEY	ACTION
	Move one position left. Ignored if already at the left margin.
	Move one position right. Ignored if already at the right end of the line.
	Move one position up. If the line above is shorter than the current column position, move left to the end of that line. Scroll down one line if the cursor is on the top line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.
	Move one position down. If the line below is shorter than the current column position, move left to the end of that line. Scroll up one line if the cursor is on the bottom line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.

Table 4-1. Edit Mode Keys (cont)

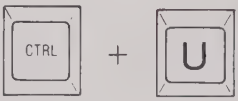
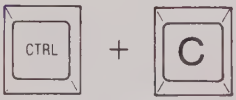
KEY	ACTION
	Delete from the cursor position to the end of the line. If the cursor is at the left margin, delete the entire line and move the rest of the program up one line to fill the deletion.
	Delete the character at the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the right of the cursor will progressively disappear.
	Delete the character to the left of the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the left of the cursor will progressively disappear as the portion to the right moves to the left margin. This key function is also available in Immediate Mode.
	Delete the current line.
	Move to the left margin.

Table 4-1. Edit Mode Keys (cont)

KEY	ACTION
	Move to the right end of the line.
	When the cursor is at the right end of the line, open a new empty line below, and move to its left margin. When the cursor is not at the right end of the line, break the line into two lines. The cursor position identifies the first character of the new (second) line. This action will not be done if the portion of the line that was to the left of the cursor does not pass a syntax check.
Character Keys	Insert characters at the current cursor position. Each character entry moves the cursor right one position along with any text to the right of the cursor. Entries that would result in a line length greater than 79 characters are not accepted, and produce a beep sound.
	Return to Immediate Mode.

Additional Editor Features

It is not necessary to insert a line in correct sequence. Regardless of the order in which program lines are entered, the editor will store them in memory in the correct line number sequence. When the cursor is instructed to move from a line, the editor checks for some syntax errors, such as omitting a quote, parenthesis, or line number. If a line does not pass the check, an error message is displayed and the cursor is not allowed to leave the line until the error is corrected.

If a program line is renumbered by deleting all or part of its line number and then entering a new line number, a duplicate line will result. One line will have the original line number, the other line will have the new line number. This may be seen by scrolling the modified line on and off the display, in EDIT mode. This can be convenient for duplicating sections of programs.

If ⟨CTRL⟩/C is entered when the current line will not pass the syntax check, the blinking error message is displayed in Immediate Mode and the line is not stored in memory.

There are many errors the editor will not detect, such as forgetting to dimension an array or specifying GOTO with a nonexistent line number. Such errors will be detected only when the program is run.

The cursor will not scroll above the lowest line number nor below the highest line number in the program. If the cursor is in the middle of the program and a new last line is entered at that position, the cursor will not scroll down past that line. To correct this condition:

- The cursor may be scrolled in the opposite direction until the line entered out of sequence disappears from the display. Reverse scroll direction again and the line will then be in proper sequence.

- Type ⟨CTRL⟩/C, and then type EDIT, followed by the line number that needed editing. Lines will then be displayed in correct sequence, allowing access to all lines.

The editor stores program lines in memory with the line number shown on the display. If any other program line has the same line number as that shown on the display, it is replaced with the contents of the line shown on the display. This feature can be used to duplicate program lines by changing only the line number and moving the cursor off the line. The line with the previous line number is not deleted by this process. The display, however, will show only the most recent line number entered. To see both resulting lines, scroll the entered line off the display and back on.

When a line is scrolled off the display with the same line number as a line previously stored, the original line in memory will be replaced by the one which is scrolled off. In order to prevent this from occurring, assign each line a unique line number.

Section 5
Storing and Running a Program

CONTENTS

Introduction 5-3
Overview 5-3
Saving a Program 5-4
SAVE, RESAVE, SAVEL, and RESAVEL Immediate Mode Commands..... 5-4
Running a Program 5-6
OLD Immediate Mode Command 5-6
RUN Immediate Mode Command 5-8

INTRODUCTION

After writing a program and having it execute (run) successfully, most users want to preserve the fruits of their labor, either for history or for later use. This section describes how to save a program on one of the file-structured devices (floppy disk or Electronic disk, for example), load a previously saved program into memory, and how to load and run a previously saved program.

OVERVIEW

The procedures discussed in the last section produce a program in main memory. However, whenever the power is turned off or, the user exits the BASIC interpreter program to FDOS (using EXIT or ⟨CTRL⟩/P), the program in memory is deleted. To save a program for later use, it should be stored on a floppy disk.

Programs may also be stored in main memory on the Electronic disk. This requires that a portion of main memory has been previously file-structured using the FUP /C command, prior to running the BASIC interpreter program. Electronic disk procedures are identical to those for the floppy disk. Remember, however, that the electronic disk will lose its contents whenever power is turned off.

SAVING A PROGRAM

Fluke BASIC has four statements that cause the program currently in memory to be saved on the named device or the system device if no device is specified. These statements are: SAVE, SAVEL, RESAVE, and RESAVEL.

SAVE, RESAVE, SAVEL, and RESAVEL Immediate Mode Commands

- The SAVE and RESAVE statements save the program currently in memory onto the named device, or the system device if none is specified, in ASCII form. This form is suitable for later editing with the System Editor or for printing on a hardcopy device.

- The SAVEL and RESAVEL statements save the program currently in memory onto the named device, or the system device if none is specified, in lexical form. This form is an intermediate form to the BASIC interpreter program that requires less file space and fewer processing steps to load and run. In a nutshell, use the SAVEL or RESAVEL statements to minimize the time necessary to load the program into memory and begin execution.

  NOTE

  A program saved via SAVEL or RESAVEL may not be executable (or loadable) if the version of Fluke BASIC under which it was saved differs from the version under which it is to be executed.

- A program that has been stored on a file-structured device may be edited with the BASIC Editor once it has been loaded into main memory with the OLD or RUN statements.

- The SAVE and SAVEL statements are interactive. Both statements ask the user for permission before overwriting an existing file. If the first letter in the answer is "y" or "Y" then the existing file is overwritten. Any other response aborts the SAVE or SAVEL statement.

- The RESAVE and RESAVEL statements are not interactive. Both statements write the program currently in main memory to the named device, or to the system device (if no device is specified). If the filename given already exists on the device, it is overwritten, without asking.

Example:
A program is currently in memory. To save this program on a floppy disk, with a filename of TEST.BAS follow the procedure below. For the purpose of this example, the device name (MF0:) is specified explicitly. If MF0: is designated as the System Device, then the device name specification may be omitted.

```
Ready
save "mf0:test.bas"

Ready
save
Replace existing file TEST.BAS? no

Ready
resave

Ready
```

NOTE

Once the filename is given as an argument to the OLD, RUN, or any of the SAVE commands, the filename is stored in the Controller's memory. If any of these commands are used subsequently, the filename is not required.

RUNNING A PROGRAM

BASIC provides two ways to retrieve a program from a floppy disk or any other file-structured device. The OLD command loads a program into memory and remains in Immediate Mode. The RUN command loads a program into memory and immediately transfers control to the program and places BASIC into the Run Mode.

OLD Immediate Mode Command

The OLD command is used to load a program into memory from a file-structured device.

- The file name, including optional storage device prefix and name extension, must be enclosed in quotes.
- BASIC will look for the file on the System Device if the a device is not specified as a file name prefix. Refer to the Input and Output section for a discussion of the System Device.
- BASIC will look for the file on a specified device if the device name is included as a file name prefix (MF0: for the floppy disk, and ED0: for the electronic disk).
- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form. A discussion of lexical form is included under SAVEL in the Reference Section of this manual.
- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.
- If no extension is specified, BASIC looks for a file with .BAL name extension and loads that file if it exists.
- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.
- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command directly specifies otherwise.
- The following two examples illustrate two ways of using the OLD Immediate Mode Command.

```
Ready
old "test"

Ready
```

Load the file named TEST.BAL (if present) TEST.BAS (if TEST.BAL is not present) from the default System Device into memory.

```
Ready
old "mf0:test.5"

Ready
```

Load the file named TEST.5 from the floppy disk.

RUN Immediate Mode Command

The RUN command loads the named file from the named device and immediately begins running the program just loaded. If no device is specified, the System device (SY0:) is used. If no other file is specified, the program in memory is run.

- The file name, including optional storage device prefix and name extension, must be enclosed in quotes.
- BASIC will look for the file on the default System Device if a device is not specified as a file name prefix. Refer Section 3 of the System Guide for a discussion of the System Device. File names are discussed in Section 1 and Section 2 of this manual.
- BASIC will look for the file on the specified device if the device name is included as a file name prefix (MF0: for the floppy disk, and ED0: for the electronic disk).
- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form. A discussion of lexical form is included under SAVEL in the Reference volume of this manual set.
- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.
- If no extension is specified, BASIC looks for a file with .BAL name extension and loads that file if it exists.
- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.
- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command directly specifies otherwise.
- The RUN command may also be used in RUN mode to chain (sequentially run) another BASIC program. This is described later in this manual and in the Reference Section.

Section 6
Data, Operators, and Expressions

CONTENTS

Introduction 6-3
Overview 6-3
Data Types 6-4
  Floating-Point Data 6-4
  Floating-Point Constants 6-5
  Integer Data 6-5
  Integer Constants 6-6
  String Data 6-6
  String Constants 6-7
  System Constants 6-7
BASIC Variables 6-8
  Introduction To Variables 6-8
  Floating Point Variables 6-8
  Integer Variables 6-8
  String Variables 6-8
  Array Variables 6-9
  System Variables 6-11
Operators and Expressions 6-12
  Assignment Statement: = ... 6-12
  Arithmetic Operators: + - * / 6-12
  Relational Operators: = < > <> <= >= 6-14
    Numeric Comparisons 6-15
  String Concatenation Operator: + ... 6-15
  Logical (Bitwise) Operators 6-16
    Binary Numbers 6-16
    Twos Complement Binary Numbers 6-18
    AND Operator 6-18

OR Operator 6-19
XOR Operator 6-19
NOT Operator 6-20
Operator Hierarchy 6-20
Use of Parentheses in Expressions 6-22

INTRODUCTION

Data and Operators are the building blocks of a BASIC expression. Data are the numbers and strings used in an expression. Operators are used to perform various fundamental arithmetic and logical operations. Expressions combine data and operators into valid BASIC program lines.

OVERVIEW

BASIC is designed with the ability to compute with numbers and strings. This section covers the data, data types and operators used by the programmer to build expressions. These expressions are a precise list of directions which BASIC follows to compute a result.

DATA TYPES

Floating-Point Data

Floating-point data has the following characteristics:

- Decimal exponent range: +308 to –308.
- Exact range: 2.2225074E–308 to 3.595386E+308.
- Resolution: 15 decimal digits.
- Inexactness in the numeric representation. This inexactness can cause problems when comparing two floating-point numbers.
- Memory requirement (per data item): 8 bytes.
- Represented internally in binary in accordance with proposed standard "IEEE Floating-Point Arithmetic for Microprocessors". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York, 10017.

Unless modified by a PRINT USING statement, floating-point data is displayed with a leading space or sign and a trailing space. It is printed out to seven significant digits. A value from .1 to 9999999 inclusive is printed out directly. A number less than .1 is printed out without E notation if all of its significant digits can be printed. All other values are printed in E notation (+0.dxxxxxxE+yyy), where d is a non-zero digit, x is any digit, and trailing zeros are dropped.

Floating-Point Constants

Floating-point constants, often called real numbers, are represented in a program in decimal or possibly scientific notation. The syntax diagram illustrates the proper representation of floating-point numbers. A number in scientific notation, with an exponent following "E", represents a number multiplied by a power of 10. Examples of floating-point constants are:

.005	
6354.33	
-134.7	
-12E2	Represents -1200
0.13E-05	Represents .0000013
0.1E6	Represents 100000
-.1E-400	Floating-point number outside the legal range. Returns error 602.

NOTE

The inexactness in representation of floating-point numbers can cause problems when using the equality operator (=) to compare two values. To check equality of floating-point numbers, compare the absolute value of their difference to a small enough limit. For example, use ABS(A - B) < 1E-15 instead of A = B.

Integer Data

Integer data has the following characteristics:

- Range: -32768 to +32767
- Resolution: Integers
- Exactness.
- Memory requirement (per datum): 2 bytes
- Integer data is represented internally in binary but displayed by the Instrument Controller in decimal without the modification process described in Floating-Point Data.
- Operations that call for an integer result are rounded to an integer, if necessary.

Integer Constants

Integer constants are whole numbers identified by a "%" suffix on the number.

Examples of integers are:

–0%
5%
–32000%
–40000% Outside the allowed range.

String Data

Strings are sequences of 8-bit positive integers that are normally interpreted as ASCII characters. Strings are used to store characters for messages to instruments and to the display, as well as for storage of binary data taken from instruments. String data has the following characteristics:

- Maximum length limited only by available memory or 16383 characters.
- Memory requirement: each string of 16 or less characters occupies an 18-byte memory segment and an additional 18 bytes for each additional 16 characters.
- String data is normally displayed by the Instrument Controller in ASCII. See the Touch Sensitive Display section for exceptions.
- When interpreted as ASCII, the value of the most significant (8th) bit is ignored. See Appendix G, ASCII/IEEE-488 Bus Codes.

String Constants

String constants are expressed as a sequence of printable characters (numerics, uppercase alphabetic characters, lowercase alphabetic characters, printable symbols, e.g., *, –, [, etc). In most cases, string constants must be enclosed in either single or double quotes. Enclosing the statement in single quotes allows the use of double quotes in the constant and vice versa. String constants need not be expressed in quotes when part of a DATA statement or when entered after an INPUT statement. Some examples of strings are:

```
A$ = "The result of 3.8 * PI is "
IN$ = 'Reply with "YES" or "NO" '
```

System Constants

Fluke BASIC makes available two floating-point constants. Under the name PI is stored the value 3.14159265358979. The mathematical function EXP(X) computes the result of raising the base (e) of the natural logarithm to a power expressed by the value of X. The function EXP(1) produces the stored value of e, 2.71828182845905. Refer to the Functions section for further information.

BASIC VARIABLES

Introduction To Variables

Variables are named data items that may be changed by the actions of a running program.

- They may be defined by the program itself (user variables), or they may be defined by the Instrument Controller operating software (system variables).
- User variables are assigned a value by the assignment statement, by the READ statement, or by an INPUT statement.
- An assignment statement assigns a value (the result of evaluating an expression), to a variable. One form of an expression is a constant. For example, A=2.
- The READ statement and statements which input data associate a variable name with a constant.
- System variables store changing-event information, such as time of day or length of last file opened, for use as required by a program.

Floating-Point Variables

Floating-point variables are designated by a letter followed by an optional second character. The second character can be a letter or a number. The following variable names are not allowed, since they are keywords of Fluke BASIC: AS, FN, IF, LN, ON, OR, PI, TO.

Integer Variables

Integer variables are designated by a floating-point variable name followed by a "%" character.

String Variables

String variables are designated by a floating-point variable name followed by a "$" character.

Array Variables

An array variable is a collection of variable data under one name.

- Arrays consist of floating-point, integer, or string variables.
- The variable name has either one or two subscripts to identify individual items within the array.
- Subscripts are enclosed in parentheses.
- When two subscripts are used, they are separated by a comma.
- It is helpful to view two-dimensional arrays as a matrix. The first subscript is the row number, and the second subscript is the column number. For example, FT%(3,18) identifies the integer in row 3, column 18 of the array FT%(m%,n%).
- A subrange (portion) of an array can be designated by specifying a first and last subscript separated by two periods.

For Example:

A$(3..7)	Strings 3 through 7 of the string array A$.
FT%(2..4, 5..15)	Rows 2 through 4 in columns 5 through 15 of the integer array FT%.

- In the last example above, the second subscript is incremented or decremented before the first. For example, the statement PRINT FT%(2..4,5..15) will display the range FT%(2, 5 through 15) before displaying FT%(3, 5 through 15).
- Array variables are distinct from simple variables. A and A(0) are two different variables.
- Only one array variable can be associated with an identifier. A%(n) and A%(m,c) are not simultaneously allowed.
- Memory space must be reserved for an array variable before it can be used. See the discussion of the DIM statement in Section 7 of this manual.

- Virtual arrays are array variables accessible through a channel to the floppy disk or the optional electronic disk. This feature allows a program to take advantage of the much greater storage space available on these mass storage devices. Refer to the Section 7 of this manual.

- Some examples of array variables are:

```
A%(3)
B1%(2%, 3%)
A$(5)
C(3%)
D(2 + A * B, C)
D( D(2) )
```

System Variables

System variables store changing event information for use as required by a program. They are accessed by name and return a result in floating-point, integer, or string form as appropriate. The system variables in Fluke BASIC are summarized in Table 6-1.

Table 6-1. System Variables

NAME	TYPE	EXAMPLE	MEANING
CMDLINE$	String	basic	Command line as typed to the FDOS⟩ prompt.
DATE$	String	08-Feb-83	Current date in the format DD-MM-YY.
ERL	Integer	1120	Line number at which the most recent BASIC program error occurred.
ERR	Integer	305	Error code of the most recent error the BASIC interpreter found in the program being executed.
FLEN	Integer	6	Length of the last file opened in 512-byte blocks.
KEY	Integer	20	Position number of the last Touch Sensitive Display region pressed.
MEM	Floating-Point	29302	Amount of unused main memory, in bytes.
RND	Floating-Point	0.2874767	Pseudo-random number greater than 0 and less than 1.
STIME$	String	14:03:06	Current time of day in 24-hour format, including seconds.
TIME	Floating-Point	0.5491405E+08	Number of milliseconds since the previous midnight.
TIME$	String	17:45	Current time of day in 24-hour format.

NOTE: Accessing the KEY system variable resets its value to zero after access.

OPERATORS AND EXPRESSIONS

Assignment Statement: =

When used as a program statement, with a variable to its left, and an expression to its right, = assigns the result of evaluating the expression to the variable. No test of equality is implied. This is the default form of the LET statement. In the following example, the integer N% is incremented by 1.

```
N% = N% + 1%
```

Arithmetic Operators: + - * /

Arithmetic operators act upon or between numbers or numeric expressions to produce a numeric result. A numeric expression is composed of an arithmetic operator and one or more operands. Table 6-2 summarizes the arithmetic operators. The following guidelines apply:

- The + and - operators can act upon a single number or numeric expression (unary operation).
- All other arithmetic operators act between two numbers or numeric expressions.
- Numeric variables can be used as numbers in expressions if they have previously been assigned a value.
- Floating-point numbers and integers may be intermixed. Integers will automatically be converted to floating-point, if necessary.
- When one integer is divided by another, the result is a truncated integer quotient (the fraction or remainder is truncated). For example:

  ```
  2% / 5% is 0%
  15% / 3% is 5%
  17% / (-3%) is -5%.
  ```

- When a result is assigned to a numeric variable, it is first automatically converted to the data type of the numeric variable.
- When a floating-point number is assigned to an integer variable it is rounded to an integer value, not truncated.

- Computation speed is significantly faster when floating-point and integer data types are not intermixed.
- The result of evaluating an arithmetic expression may be used in a larger expression or assigned to a variable for later use.
- Except for +, arithmetic operators cannot be used on strings.

Table 6-2. Arithmetic Operators

OPERATOR	NAME	MEANING AND EXAMPLES
+	Positive	Unary plus operator. Does not change sign.
+	Add	Add two numeric quantities.
—	Negative	Unary minus operator. Changes the sign.
—	subtract	Subtract two numeric quantities.
*	Multiply	Multiply two numeric quantities.
/	Divide	Divide first numeric quantity (dividend) by the second numeric quantity (divisor) to produce a quotient.
^	Exponentiate	Raise the first numeric quantity to a power equal to the second numeric quantity.

Relational Operators: = < > <> <= >=

Relational operators compare numeric values or character strings. A relational expression returns an integer Boolean result of 0 for false, and –1 for true. The structure of a statement determines whether the = operator is used for a relational comparison or an assignment. Two examples are as follows:

```
PRINT A = B   Displays –1 if A and B are equal.
A = B = C     Assigns –1 to A if B and C are equal.
```

Table 6-3 lists and describes the relational operators in Fluke BASIC. All of the relational operators are described in greater detail in the Reference volume of this manual set.

Table 6-3. Relational Operators

OPERATOR	ACTION
=	Relational equality operator.
<	Relational less than operator.
<=	Relational less than or equal to operator.
>	Relational greater than operator.
>=	Relational greater than or equal to operator.
<>	Relational not equal operator.

Numeric Comparisons

Numeric comparisons are made as follows:

- All negative numbers are "less than" zero or any positive number.
- Integers are converted to floating-point when a comparison is between mixed numeric data types. This conversion requires additional processing time.
- When an operator checks for equality or inequality of numeric expressions, use integers wherever possible. This is due to the inexactness, and rounding and truncation errors, of floating-point values.
- To check equality of floating-point numbers, compare the absolute value of their difference to a small enough limit. For example, use ABS(A – B) ⟨1E-15 instead of A = B.

String Concatenation Operator: +

When used between character strings, + concatenates (connects) the strings together. The following examples illustrate the use of this operator.

EXPRESSION(S)	RESULT
"BEGIN" + "OPERATION"	BEGINOPERATION
"BEGIN " + "OPERATION"	BEGIN OPERATION
A$ = " Volts" "ENTER" + A$	ENTER Volts
A$ = "Millivolts" "ENTER" + A$	ENTERMillivolts

Logical (Bitwise) Operators

Logical operators AND, OR, XOR, NOT, operate on the binary digits that make up an integer. NOT is a unary operator, acting upon one integer. AND, OR, and XOR are binary operators, using the bits of one integer to act upon another integer. Logical operators allow examination or modification of integer bit patterns when they have been used to store binary data, such as status or binary readings from instrumentation.

Binary Numbers

To use logical operators effectively, it is necessary to understand the binary number system, and how the Instrument Controller uses binary numbers to represent integers.

The Instrument Controller uses a 16-bit (2-byte) word to store each integer. Each bit position represents a weighted power of 2. The sum of the weight column in the following chart is the number of separate integers that can be represented with 16 bits less one because zero is an additional integer.

BIT POSITION	WEIGHT
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
SUM:	65535

Integer numbers are represented by setting appropriate bit positions to 1. Adding the weighted values of each position that is set to 1 gives the decimal value of the integer. For example, the number 305 is represented by the binary pattern 0000 0001 0011 0001, as follows:

```
Position: 15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0

Setting:  0  0  0  0   0  0 0 1  0 0 1 1  0 0 0 1
```

Since each bit that is set to 1 carries a binary weight, it is possible to verify that the binary pattern is correct. adding the numbers 56 + 32 + 16 + 1 gives the decimal value 305.

Decimal numbers can be converted to binary by continuously dividing by 2 and keeping track of the remainders (always 1 or 0). For example, the number 305 is converted to binary as follows:

```
305 / 2 = 152    R = 1
152 / 2 = 76     R = 0
 76 / 2 = 38     R = 0
 38 / 2 = 19     R = 0
 19 / 2 = 9      R = 1
  9 / 2 = 4      R = 1
  4 / 2 = 2      R = 0
  2 / 2 = 1      R = 0
  1 / 2 = 0      R = 1
```

Reading the remainder bits, from the bottom up, gives the result 100110001, the binary representation of 305.

Twos Complement Binary Numbers

By using the most significant bit (15) to identify a negative integer, the Instrument Controller divides the pattern into 32767 positive numbers, the number 0, and 32768 negative numbers. Negative numbers are represented in a form called twos complement. To change either to or from twos complement form, the following steps are required:

1. Replace every 0 with a 1.
2. Replace every 1 with a 0.
3. Add 1.

For example, to change the pattern 0000 0001 0011 0001 (+305) to twos complement form, first reverse 1's and 0's: 1111 1110 1100 1110, and then add 1: 1111 1110 1100 1111 (–305). To change it back, first reverse 1's and 0's: 0000 0001 0011 0000, and then add 1: 0000 0001 0011 0001.

AND Operator

AND returns an integer bit pattern with a 1-bit in every position where both of two input integers have a 1-bit. The AND operator is useful to check for the setting of particular bit(s) to 1 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 1. The following examples illustrate the results of AND operations:

33% AND 305%	33% 0000 0000 0010 0001 305% 0000 0001 0011 0001
RESULT: 33%	0000 0000 0010 0001
–74% AND 305%	–74% 1111 1111 1011 0110 305% 0000 0001 0011 0001
RESULT: 304%	0000 0001 0011 0000

OR Operator

OR returns an integer bit pattern with a 1-bit in every position where either of two input integers have a 1-bit. The OR operator can be used to check for the resetting of particular bit(s) to 0 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 0. The following examples illustrate the results of OR operations:

```
33% OR 305%      33% 0000 0000 0010 0001
                305% 0000 0001 0011 0001
                ---------------------
RESULT: 305%        0000 0001 0011 0001


-74% OR 305%    -74% 1111 1111 1011 0110
                305% 0000 0001 0011 0001
                ---------------------
RESULT: -73%        1111 1111 1011 0111
```

XOR Operator

XOR (Exclusive OR) returns an integer bit pattern with a 1-bit in every position where the bits of two input integers are opposite. A mask word applied to an unknown integer through XOR will invert (1 to 0, and 0 to 1) all bit positions where the mask contains a 1, and leave unchanged all bit positions where the mask contains a 0. The following examples illustrate the results of XOR operations:

```
33% XOR -1%      33% 0000 0000 0010 0001
                 -1% 1111 1111 1111 1111
                ---------------------
RESULT: -34%        1111 1111 1101 1110


-74% XOR 0%     -74% 1111 1111 1011 0110
                  0% 0000 0000 0000 0000
                ---------------------
RESULT: -74%        1111 1111 1011 0110
```

NOT Operator

NOT is a unary operator that operates upon a single integer. NOT returns a 1 bit in every position where the input integer had a 0 bit, and a 0 bit in every position where the input integer had a 1 bit. The following examples illustrate the results of NOT operations:

```
NOT 33%          33% 0000 0000 0010 0001
                     ---------------------
RESULT: -34%         1111 1111 1101 1110


NOT -74%        -74% 1111 1111 1011 0110
                     ---------------------
RESULT: 73%          0000 0000 0100 1001
```

Operator Hierarchy

As long as the results of individual operations are compatible with each other, operators can be combined in any order within an expression. However, BASIC follows certain internal rules of hierarchy when evaluating expressions. Table 6-4 lists the operators discussed within this section in seven levels of hierarchy.

- An expression is scanned left to right for level 1 operations (exponentiation).
- After performing these, the expression is scanned left to right for level 2 operations (+ sign, – sign).
- This sequence is continued until level 7 operations have been scanned and performed, if present.
- Within the same priority level, operations are performed in left to right sequence.
- This sequence can be modified by the use of parentheses.

The following example illustrates these concepts:

The expression: 1 + 2 * 7 4 / 3 - 5

is evaluated as (7 4) * (2 / 3) + 1 - 5 (hierarchal order)

not as ((((1 + 2) * 7) 4) / 3) - 5 (left-to-right order)

Table 6-4. Operator Priority

PRIORITY	OPERATOR	FUNCTION
1	^	Exponentiate
2	+	Positive Sign (Unary plus)
2	—	Negative Sign (Unary minus)
3	*	Multiply
3	/	Divide
4	+	Add
4	—	Subtract
5	<	Less Than
5	>	Greater Than
5	=	Equals
5	< >	Not Equal To
5	<=	Less Than or Equal To
5	>=	Greater Than or Equal To

Use of Parentheses in Expressions

After the rules of operator hierarchy are satisfied, expression evaluation normally proceeds from left to right. Parentheses can be used to organize and change the evaluation sequence of expressions. The following rules govern the interpretation of parentheses by Fluke BASIC:

- Parentheses have priority over all operators.
- Each left parenthesis "(" must have a corresponding right parenthesis ")" i.e., parentheses must appear in pairs.
- A pair of parentheses may be nested within another pair. Line length is the only limit to this nesting.
- Evaluation of nested parentheses proceeds from the innermost pair outwards.
- Pairs of parentheses that are not nested within each other are evaluated in left to right sequence.
- Parentheses may also be used where they have no effect except clarity to the programmer.

The following example illustrates these concepts. The variables have the values A = 2, B = 3, C = 4, D = 5, and E = 6.

The expression A + B * C + D / E will evaluate as:

```
2 + 3 * 4 + 5 / 6
2 + 12 + .833333....
14.83333....
```

The expression (A + B) * (C + D) / E will evaluate as:

```
(A + B) * (C + D)/E
(5) * (9)/6
45/6
7.5
```

. . a very different result.

Section 7
Data Storage

CONTENTS

Introduction 7-3
Overview 7-3
Using Arrays for Data Storage 7-4
  Array Types and Differences 7-5
  Advantages and Disadvantages 7-6
I/O Channels 7-7
  Channel Input and Output 7-7
  OPENing an Output Channel 7-8
  OPENing an Input Channel 7-8
  Close a Channel 7-8
  Specifying File SIZE 7-9
Sequential Data Files 7-10
  Creating a Sequential File 7-10
  Reading a Sequential File 7-11
Creating Main Memory Arrays 7-13
Using Main Memory Arrays 7-14
  Using Main Memory Arrays as Ordinary Variables 7-14
  Programming Techniques 7-15
  Two Dimensional Arrays 7-16
  Multiple Arrays 7-17
  Redimensioning 7-17
  Main Memory Arrays and Program Chaining 7-18
  Serial Storage of Main Memory Arrays in Mass Storage . 7-19
    Device Storage Size Requirements 7-20
    Device File Size Calculation 7-20
    Disk Size Calculation Example 7-21

Virtual Array Files 7-22
Definition 7-24
Advantages and Disadvantages 7-24
File Structure 7-26
Analogy 7-26
Virtual Array File Organization 7-27
Storage Size Requirements 7-29
Virtual Array Size Calculation 7-30
Creating Virtual Arrays 7-32
Using Virtual Arrays 7-33
Using Virtual Arrays as Ordinary Variables 7-33
Using Virtual Array Strings 7-34
Programming Techniques 7-35
Array Element Access 7-35
Splitting Arrays Among Files 7-36
Reusing Virtual Array Declarations 7-37
Equivalent Virtual Arrays 7-38

INTRODUCTION

Many program applications require data to be stored or retrieved from some form of long term storage. For instance, a digital voltmeter connected to the IEEE-488 bus does not have the capability to average its readings over a long period of time. The solution is to take ten (or however many) readings, store them, then use the math capability of the Controller to compute the average.

Using simple variables to store the readings is effective, but addressing these variables systematically in a program can be clumsy. A better technique is to store the data in an array. This allows addressing via the array subscript, which can be a numeric expression.

OVERVIEW

This section describes the techniques of data storage, both in main memory and on file-structured devices. After discussing the advantages and disadvantages of both, the methods of use and the statements used are discussed.

USING ARRAYS FOR DATA STORAGE

An array is both a powerful and convenient method of storing a large volume of data. There are 954 possible names for each type of simple variable in Fluke BASIC. This may seem like a large number, but it is not enough to store the readings from a typical digital voltmeter measurement. Consider the problem of storing 5 readings using sequential, but independent variables using a loop.

```
10 FOR X = 1 TO 5
20   GOSUB 100          ! subroutine to request data (y%)
30   IF X = 1 THEN A1% = Y%
40   IF X = 2 THEN A2% = Y%
50   IF X = 3 THEN A3% = Y%
...
80 NEXT X
90 END
100 ! subroutine to get reading from dvm
110 ! more code, return value in y%
120 RETURN
```

This program fragment will work, but it is easy to see how involved and clumsy it could get, especially if a large number of data items were involved. Notice that each reading consumes a program line and one variable name. Here is a program fragment using an array to accomplish the same thing.

```
10 DIM A%(6%)
20 FOR X = 1 TO 5
30 GOSUB 100 ! go get reading (y%)
40 A%(X%) = Y%
50 NEXT X
60 END
100 ! subroutine to get reading from dvm
110 ! more code, return value in y%
120 RETURN
```

In line 40, the array element is assigned the value returned from the subroutine (Y%) at line 100. The next iteration through the loop increments the value of X%, which also causes line 40 to assign Y% to the next sequential array element.

If the number of readings to be made and stored were increased to 1000, the only program changes needed would be to lines 10 and 20. In the first example 995 more program lines would need to be added inside of the FOR-NEXT loop.

Array Types and Differences

Fluke BASIC allows data values to be stored by one of the following methods.

- As a sequentially-accessed data file on a file-structured device. This is essentially a list of data items.
- In main memory. Main memory arrays may be one or two-dimensional.
- As a virtual array (random access data file) on a file-structured device. The array may be one or two-dimensional.

Advantages and Disadvantages

Each of the three storage methods has its strong and weak points. These are summarized below.

- The sequential data file and virtual array are both stored on file-structured devices. If the device is a non-volatile file-structured device (such as the floppy disk or bubble memory), the array storage is also non-volatile.

- A main memory array, being stored in main memory (system RAM), is susceptible to power failures, program chaining and DELETE ALL statements.

- A sequential data file must be accessed sequentially. If you want the 405th element of the file, you must sequentially access the preceeding 404 elements first, which can be time consuming.

- Sequential data files must be written, then read. A sequential file may not be simultaneous read/write.

- Virtual arrays and main memory arrays are random access. Any array element may be accessed at any time.

- An array stored in main memory can have a shorter access time than the same array stored on a file-structured device.

- All three storage methods will accept integer, string or floating-point (real) values.

- The elements of a virtual string array have a definite, dimensioned length. Elements of a main memory string array are limited in length only by the amount of main memory available.

- Virtual arrays can have up to 65,536 (64K) bytes of data (128 blocks).

- Main memory arrays are limited to 28K bytes minus the program size in K bytes. Assuming a 5K application program, this means a 23K maximum array size.

- Virtual arrays and main memory arrays can be used like ordinary program variables.

I/O CHANNELS

The Instrument Controller communicates between the BASIC program and various devices by means of I/O channels.

The *devices* used for data storage are *file-structured* devices such as:

MF0: (floppy disk)
ED0: (E-disk)
WD0: (fixed disk drive)
MB0: (bubble memory)

There can be a maximum of 16 I/O channels open at any one time. They are designated with the numbers 1 through 16.

The I/O channels are also used to communicate with instruments connected to the IEEE-488 bus and RS-232 devices. Refer to sections 9 and 10 of this manual.

Channel Input and Output

The BASIC statements used in conjunction with the I/O channels are the PRINT statement and the INPUT statement.

- The PRINT statement is used to output data from the program to the device via the I/O channel.
- The INPUT statement is used to input data from the device, via the I/O channel to the program.
- The #n clause is added to both the PRINT and INPUT statements when they are used for channel I/O. The #n clause specifies the channel number to be used for input or output.
- The PRINT and INPUT statements are fully described in the Reference Section of this manual.

OPENing an Output Channel

To designate an output channel, it is necessary to OPEN it AS a NEW FILE. When an output channel is to be used for a virtual array, the DIM clause is used in the OPEN statement.

Example

```
10   OPEN "MFO:EXAMPL.DAT" AS NEW FILE 1    ! new file, channel 1
20   OPEN "EDO:TEST.DAT" AS NEW FILE 2      ! new file, channel 2
30   OPEN "MFO:TEST.VRT" AS NEW DIM FILE 3 ! virtual array, chan 3
30   PRINT #1, "HELLO"                      ! output "HELLO" to channel 1
40   PRINT #2, A$                           ! output A$ to channel 2
```

NOTE

A virtual array also requires the use of the DIM statement. Refer to the discussion of Virtual Arrays, elsewhere in this section.

OPENing an Input Channel

To designate an input channel, it is necessary to OPEN it AS an OLD FILE.

Example

```
10   OPEN "MFO:EXAMPL.DAT" AS OLD FILE 3 !input from channel 3
20   OPEN "EDO:TEST.DAT" AS OLD FILE 4   ! input from channel 4
30   INPUT #3, A$                        ! read first line, chan 3
40   INPUT #4, B$                        ! read first line, chan 4
```

An error will result if an attempt is made to OPEN a channel that has previously been OPENed but not CLOSEd. It is a good practice to close channels at the end of a program unless files are passed to a chained program.

CLOSE a Channel

The CLOSE statement closes the file associated with the channel number given. It is a good idea to CLOSE a channel immediately prior to OPENing that channel.

Example

```
10 CLOSE 1
20 CLOSE 2
```

Specifying File SIZE

If no file size is specified, the largest contiguous file space will be temporarily reserved for that file. This may not leave any device space available for additional files. To overcome this, specify the SIZE in blocks that you wish reserved on the device for that file. Methods for calculating block size are described in the discussions of Main Memory Arrays and Virtual Arrays.

Example

```
20 OPEN "MF0:FILE3.DAT" AS NEW FILE 1 SIZE 2
30 OPEN "MF0:FILE4.DAT" AS NEW FILE 2 SIZE 8
```

SEQUENTIAL DATA FILES

A sequential data file is a list of data items, separated by ⟨CR⟩⟨LF⟩. As its name implies, a sequential file must be read sequentially; there is no means to access an item within the file except by reading all the items that preceed it.

Sequential files store data in ASCII format and may be read using the COPY statement or FUP. To copy a sequential file to the screen, type the following command from BASIC:

COPY {pathname}

or, from FDOS:

FUP {pathname}

Creating a Sequential File

Creating a sequential file is best illustrated with a short program. The discussion that follows describes the important points to remember.

```
10   CLOSE 1    ! be sure channel is closed before opening
20   Y% = 1%
30   OPEN "TEST.DAT" AS NEW FILE 1% !test.dat, new file, chan 1
40   FOR X% = 1 TO 5%
50   PRINT #1, Y%     ! output y% to channel 1
60   Y% = 2 * Y%      ! do something to y%
70   NEXT X%
80   CLOSE 1
90   END
```

The example program opens a new file named "TEST.DAT", then writes the value of Y% to the file. The FOR-NEXT loop causes this to repeat 5 times.

Line 10 closes the channel to avoid a possible error if the channel had been opened previously and not closed.

Line 30 opens the file "TEST.DAT" and assigns channel 1 for data transfers to the file. The NEW clause tells BASIC to open a NEW file to store data. If the file "TEST.DAT" exists already, it is overwritten.

The PRINT statement within the FOR-NEXT loop sends the value of Y% to channel 1 (and hence to the file).

Line 80 CLOSEs channel 1 (good housekeeping).

Reading a Sequential File

Reading a sequential file is illustrated here with three examples. The discussion that follows each program, and the comments within the programs emphasize the important points.

Example 1

The COPY statement provides a very simple method of reading a sequential file. It is so simple, however, that it is not possible to direct the output anywhere else but to the display.

```
10 COPY "TEST.DAT"
```

Example 2

```
10   CLOSE 1    ! insurance
20   ! look at the file generated previously
30   OPEN "TEST.DAT" AS OLD FILE 1
40   ON ERROR GOTO 100     ! use this to detect end-of-file
50   INPUT #1, A%          ! load a% with data from file
60   PRINT A%              ! show it!
70   GOTO 50               ! repeat
100  IF ERR = 307 THEN 110 ELSE 130 ! err 307 = eof
110  PRINT "END OF FILE REACHED"
120  GOTO 150
130  PRINT "ERROR "; ERR; "OCCURED" ! other error than eof
150  CLOSE 1 \ END
```

The example program opens the file created by the previous example program, then reads each line of the file, prints the value, then loops until the end-of-file is reached. When the end-of-file is reached, the error handler at line 100 prints an appropriate message.

Line 10 ensures that the channel to be used in line 30 is not already open. An error will result if it is.

Line 30 opens the file "TEST.DAT" for reading (OLD clause) and assigns it to channel 1 for data transfer.

Line 50 assigns the first line of the file to a%. Line 60 prints a% on the display.

Line 70 causes an infinite (until an error) loop.

The error handler at line 100 prints the message at line 110 if the error was caused by reaching the end-of-file, otherwise a brief (and somewhat cryptic) error message is printed.

Line 150 closes the previously opened channel/file and exits.

Example 3

The INCHAR function may also be used to read data from a sequential data file. The INCHAR function is described in the Reference volume of this manual set.

```
10 CLOSE 1                                  ! insurance
20 OPEN "TEST.DAT" AS OLD FILE 1            ! open file
30 C% = INCHAR(1%)                          ! read character
40 IF C% = 26 THEN 70                       ! check for end of file
50 PRINT CHR$(C%);                          ! print it
60 GOTO 30                                  ! loop to do it again
70 CLOSE 1 \ END                            ! clean up; go home
```

Line 10 ensures that the channel has been closed before an attempt is made to open it at line 20.

Line 20 opens the file "test.dat" and specifies that it shall be read (as opposed to written) and associates it with I/O channel #1.

Line 30 uses the INCHAR function to read one character from the previously opened channel.

Line 40 checks C% for an end of file character and branches to line 70 if C% is an end of file character.

CREATING MAIN MEMORY ARRAYS

A main memory array is simple to create and use. Use the following steps to create a main memory array.

1. Dimension the array using the DIM or COM statement. The following statement dimensions a 100 element single-dimensioned integer array using N1% as the array variable. The use of the COM statement is described later in this section.

```
10 DIM N1%(100%)
```

2. Assign values to the array elements (in any sequence) by using the LET or INPUT statements.

3. Array element values may be used using the LET or PRINT (USING) statements or by using the array element as part of a numeric or logical expression.

USING MAIN MEMORY ARRAYS

The following paragraphs present some suggestions for using main memory arrays.

Using Main Memory Arrays as Ordinary Variables

Except for the required DIM (or COM) statement prior to use, a main memory array can be treated as an ordinary variable. That is, data may be stored or retrieved from any array element at any time, in any sequence. The COM statement is discussed later in this section, Section 13 of this manual, and in the Reference volume of this manual set.

- Any legal variable name may be an array variable.
- Remember to use the DIM or COM statement first.
- Do not use the DIM or COM statement twice on the same array.
- If your program crashes, data stored in a main memory array is lost after execution of a RUN or EDIT statement unless the COM statement was used to allocate memory space for the array.

In the following example, elements of A% may be used wherever integer array elements may be used.

```
10      DIM A%(255%)
```

The following example shows that data may be read from or written to the array simply by writing the array name in an expression or assigning a value to an array element.

```
305     IF A%(I%) ) 0% THEN 350
350     ! more code
430     LET A%(I%) = ABS (A%(I%)) + 2%
```

Programming Techniques

The program given in the introduction to this section illustrates the "nuts-and-bolts" of getting values into a main memory array. Getting values out of the array in a sequential fashion is largely the same:

1. Set up a FOR-NEXT loop.
2. Read the array elements one-by-one using the LET or PRINT statements.
3. Repeat step 2 as necessary.

Another method is to use array subranging. This is a special case of the array element identifier. The first example displays all elements of the array A$, which has previously been defined as having 20 elements in line format. The second example displays only elements 5 through 10 in columnar format (note the semicolon).

```
10 PRINT A$(0..19)
20 PRINT A$(5..10);
```

The example program in Section 7 of the System Guide demonstrates this method also. Look in the "Transfer Module" portion of the program. The semicolon following the array subrange suppresses the normal ⟨CR⟩⟨LF⟩ after each element, allowing display in multi-columnar format.

Two Dimensional Arrays

Until now, only one dimensional arrays have been used and discussed. It may help to think of a one-dimensional array as a one row matrix. For example, the preceeding example array, A$, represented as a matrix would be:

← Columns 0 through 19 →

ROW A$(0) A$(1) A$(2) A$(3) A$(4) A$(5) A$(6) A$(7) A$(8) A$(9) ... A$(19)

Fluke BASIC allows two-dimensional arrays. A two-dimensional array such as A$(1,4) has two subscripts in its array element identifier. This additional subscript gives the program the ability to create more elements than is possible with only one subscript. The additional subscript also gives the program the capability of building a matrix with ROWs as well as COLUMNs.

Suppose you had 3 production shifts and you wanted to store the total number of instruments produced by each shift for one week (5 days). The following program is an example of one way to do this task:

```
10   DIM PS%(2%,4%)          ! dimension a 3 X 5 array
20   READ PS%(0..2,0..4) ! read 15 values
30   !  SHIFT   DAY   DAY   DAY   DAY   DAY
40   !    1       1     2     3     4     5
50        DATA   10,   12,   9,    7,    10
60   !    2
70        DATA   9,    14,   10,   11,   11
80   !    3
90        DATA   11,   7,    6,    8,    10
100  FOR I% = 0% TO 2%
110  PRINT PS%(I%,0..4);
120  PRINT
130  NEXT I%
140  END
```

The previous program takes the data and PRINTs it in the same arrangement as the DATA statements. Note the subscript order is always (ROW,COLUMN). The matrix for this array is shown below:

← PS%(ROW NO., 0..4) →

		Col. 0	Col. 1	Col. 2	Col. 3	Col. 4
PS%	ROW 0	PS%(0,0)	PS%(0,1)	PS%(0,2)	PS%(0,3)	PS%(0,4)
(0..2	ROW 1	PS%(1,0)	PS%(1,1)	PS%(1,2)	PS%(1,3)	PS%(1,4)
Col. 0)	ROW 2	PS%(2,0)	PS%(2,1)	PS%(2,2)	PS%(2,3)	PS%(2,4)

Multiple Arrays

More than one array may be dimensioned in a program, eg,:

```
10   DIM AX(3,5), BX(6,10), A$(10,100)
20   DIM RL(1000)
30   DIM A5(10,10)
:
```

Redimensioning

BASIC programs are allowed to execute a DIM statement for each variable only once. An error will result if the program attempts to execute a specific DIM statement more than once. For this reason, DIM statements should appear early in the program and should not be included in subroutines. The only way around this is to re-RUN the program. RUN causes BASIC to forget all previously executed DIM statements.

Main Memory Arrays and Program Chaining

A main memory array must have been created with the COM statement to survive program chaining. The COM statement replaces the DIM statement for that array. The COM statement reserves variables and arrays in a common area for reference by chained programs.

- Only real (floating-point) and integer variables may be used with the COM statement.
- String variables may not be stored in the common area.
- Use a virtual array for string variables that must be accessed by chained programs.
- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.
- The contents of the common area are lost when the EXEC, EXIT, or DELETE ALL statements are executed.

For example, assume that a chained program requires the use of three floating point simple variables, an integer simple variable, a floating point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM A, B, C, FX, D(24%), TX(100%)
        .
        .
        .
1050    RUN "B"
1060    END                                  ! End of program A
```

The second program could then use:

```
10      ! Program B
20      COM L1, L2, L3, QX, K(24%), PX(100%)
        .
        .
        .
```

Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

Serial Storage of Main Memory Arrays in Mass Storage

A main memory array may be stored and retrieved from a mass storage device. Two methods are possible:

- Store the array in a sequential data file.
- Store the array in a virtual array.

The following examples illustrate how array data can be serially stored and retrieved from the disk (MF0:). You may use a different array name to retrieve data than you did to store the data; if the arrays are alike in type (integer, floating-point, or string) and dimensions.

The first example program stores the letters "A" through "F" in a main memory array, then writes that array to a sequential data file. The second example program retrieves that array from the data file, then prints the array.

- The array variable used in the second example could be any legal string variable. The data file holds only the data with no clue to the identity of the variable used to store the data.

Data Storage:

```
10   ! "EXAM1.BAS"
100  CLOSE 1                                  ! INITIALIZE
110  OPEN "EXAM1.DAT" AS NEW FILE 1 SIZE 1 ! RESERVE & NAME DISK SPACE
115  ! "NEW" in line 110 indicates new file created on the disk.
120  DIM A$(5%)                               ! DECLARE 5 ELEMENT ARRAY
130  FOR I% = 0% TO 5%
140       A$(I%) = CHR$(65% + I%)             ! ASSIGN LETTERS A --) F
150  NEXT I%
160  PRINT #1, A$(0%..5%)                     ! STORE ON DISK
170  CLOSE 1                                  ! CLOSE FILE
180  END
```

Data Retrieval:

```
10   !"EXAM2.BAS"
100  CLOSE 1                                  ! INITIALIZE
110  OPEN "EXAM1.DAT" AS OLD FILE 1           ! OPEN FILE, ASSIGN CHANNEL
115  ! "OLD" in line 110 indicates file already exists on disk
120  DIM A$(5%)
130  !
140  INPUT LINE #1, A$(0%..5%)                ! INPUT FILE DATA TO ARRAY
150  !
160  CLOSE 1                                  ! CLOSE FILE
170  PRINT A$(0%..5%)                         ! DISPLAY DATA FROM ARRAY
180  END
```

Device Storage Size Requirements

Before a main memory array can be stored, space must be reserved for it on the file-structured device. When a new file is OPENed, the largest available contiguous space on the storage medium (floppy disk or other file-structured device) is allocated for the single file, unless the SIZE is included in the OPEN statement. If two NEW files are OPENed without a SIZE statement and there is only one contiguous space available on the device, BASIC will display "? I/O error 306..." telling you there is no more room on the storage device, when the attempt is made to OPEN the second file. This will happen even though there is enough room on the disk for all the data you plan to store in each of the files

Device File Size Calculation

Size must be stated as an integer number of BLOCKS (1 BLOCK = 512 BYTES). An array file may contain more than one array. File SIZE must be large enough to equal or exceed the total number of storage bytes required by all of the REAL (floating-point), INTEGER, and STRING elements you plan to use in the array file. Array values are stored on the file-structured device as ASCII values. One byte of device storage is required for each character stored.

Device requirements for serial storage (no comma after the variable):

1 byte per significant digit or string character
1 byte for the sign (even though it may be + and not be displayed)
1 byte for decimal point (reals only)
1 byte for an included space (except with PRINT USING) for reals and integers
2 bytes for ⟨CR⟩ ⟨LF⟩
1 byte for EOF (end of file) character

Examples

```
PRINT #1, 3%                              requires 5 bytes
PRINT #1, -3%                             requires 5 bytes
PRINT #1, USING "$#",-3%                  requires 4 bytes
PRINT #1, 3.285                           requires 9 bytes
PRINT #1, -3.285                          requires 9 bytes
PRINT #1, USING "$#.###", -3.285          requires 8 bytes
PRINT #1, "12345"                         requires 7 bytes
```

Disk Size Calculation Example

Calculate the SIZE requirement for the DIM statement in the following program:

```
10   DIM S$(100),I%(100),R(100)
20   CLOSE 1
30   OPEN "TEST.DAT" AS NEW FILE 1 SIZE 6
40   FOR J% = 1% TO 100%
45   S$(I%) = "1234567890" \ I%(J%) = J% \ R(J%) = J%+PI
50   PRINT #1, S$(J%)
60   PRINT #1, USING "S###";I%(J%)
70   PRINT #1, USING "S###.##",R(J%)
80   NEXT I%
90   CLOSE 1
100  END
```

Assumptions:

1. All string elements = length of 10 characters
2. PRINT USING is utilized to ensure all reals are same length, and all integers are the same length.

	String	Integer	Real	EOF

BYTE SIZE = 100 [(10+2) + (4+2) + (7+2)] + 1
= 100 [27] + 1
= 2701

BLOCK SIZE = 2701 / 512 = 5.275391

Since partial blocks are not allowed; the next highest integer = 6 Blocks.

Note: Looping 94 times instead of 100 permits a SIZE of 5 blocks.

VIRTUAL ARRAY FILES

A virtual array provides the BASIC programmer with an easy-to-use, non-volatile means of program data storage. Once created, a virtual array variable may be treated like any other BASIC variable. A virtual array is bidirectional; you may read or write from it any time, in any sequence.

A virtual array may be assigned values from a main memory array and vice versa; virtual arrays can be used in equations with main memory arrays. Virtual arrays are different from main memory arrays in the following ways:

1. Main memory arrays reside in main memory. Virtual array elements temporarily reside in a main memory buffer of 512 bytes (1 block) per channel (file) number. They permanently reside on a file-structured storage medium. Virtual arrays may also reside on E-disk (expansion RAM memory), however, E-disk storage is volatile.

2. Main memory arrays are volatile because main memory is volatile. Virtual arrays survive providing they have been transferred from the main memory buffer to the non-volatile file-structured storage medium (CLOSEing the file ensures this).

3. Virtual arrays are not initialized by the DIM statement. Main memory arrays are assigned initial values by the DIM statement.

4. Main memory arrays are created with a DIM or a COM (common main memory for program chaining) statement (COM will not support string variables); virtual arrays are created with an OPEN and a DIM# statement.

5. Virtual arrays can be made equivalent, (two arrays can share the same area of file memory).

6. Virtual arrays do not require a COM statement in order to be accessed by a chained program. COM is not needed and cannot be used with virtual arrays. Main memory arrays require a COM statement to survive program chaining.

7. Elements of virtual array strings have a definite, dimensioned length. Elements of main memory array strings are limited in length only by the amount of main memory available.

8. Since main memory arrays must share main memory with the BASIC program, the maximum amount of main memory available for main memory arrays is limited to 28K bytes, minus the size of the user program in K bytes.

9. A virtual array file can be as large as 65,536 bytes. The total number of virtual array files is limited by the available file-structured storage.

10. Virtual arrays are stored as binary data and cannot be viewed by FUP.

11. Program execution errors automatically close virtual array files, making virtual array data inaccessible from the immediate mode, however, this data survives on the storage medium and can be retrieved by a program. Main memory arrays are accessible from the immediate mode after a program execution error (crash).

12. Virtual array data survives a re-RUN or EDIT of the program, but main memory array data is lost in both of these situations.

After reading the above comparison of virtual arrays and main memory arrays, it can be said that virtual arrays behave "virtually" the same as if they resided in main memory even though they actually reside on one of the file-structured storage media. With the exception of the OPEN, CLOSE and DIM# statements required by virtual arrays, the same identical programming code can be used interchangeably for virtual arrays or main memory arrays; ignoring for the moment the fact that virtual array strings require additional considerations in some situations due to their fixed length.

Definition

A virtual array is a collection of data stored in a random access file-structured storage device, such as the floppy disk or bubble memory. The data is stored in the Instrument Controller's internal format (binary) so that no conversion is required during input or output. After a channel has been opened, the virtual array is available to the program just like a main memory array.

Advantages and Disadvantages

Virtual arrays can be used to significantly extend the capability of a program. You will probably want to use virtual arrays exclusively except in situations where execution speed is critical.

Advantages:

1. Non-volatile ---survives power down of the Controller; survives program chaining and the DELETE ALL statement.

   NOTE

   A virtual array is non-volatile only if the file-structured device on which it is stored is non-volatile.

2. Virtual arrays do not "consume" main memory, thus more program space is available.

3. Random access means PRINT and INPUT statements are not necessary for data I/O.

4. Supports equivalencing.

5. String data, in addition to numeric data, can be accessed by chained programs.

6. Text messages can be stored on the storage medium rather than in main memory. The same text can be used as often as needed.

7. The program can be restarted after a Controller power down and returned to the exact place in the program where execution ceased (due to powering down). (See note after point #1, above.)

8. Virtual arrays can be many times larger than main memory arrays; up to 16 times (over 1024K bytes) as much data can exist in virtual arrays when a floppy disk, E-disk, or Winchester disk are used.

Disadvantages:

1. Virtual arrays are not allowed in RBYTE or WBYTE statements.

2. Virtual arrays execute slower than main memory arrays. This difference in execution speeds becomes significant when large amounts of data are being sorted, assigned or operated upon. Exact speed differences are dependent on the application and device. For example, the E-disk is almost as fast as main memory.

3. Unlike main memory arrays where the DIM statement assigns a 0 value to floating-point (real) and integer elements and an empty string, i.e. "", to string elements (note CHR$(0) ""); newly created virtual arrays contain whatever byte arrangement that exists on the storage medium where the arrays reside. To guard against bogus data entering your program, you may wish to initialize the entire array to some known value prior to storing data in it.

File Structure

Virtual arrays must reside on a specific physical space on the file-structured device in order for the program to know where it can go to store and retrieve data. An analogy to the disk file structure is a helpful aid to understanding where virtual arrays are stored.

Analogy

Imagine you have a storage room (disk) which contains 80 file cabinets (tracks) and each cabinet has 10 file drawers (blocks). Each file drawer contains 512 folders (bytes). The terms "Blocks" and "Sectors" are used interchangeably.

The file name for a specific storage space for virtual arrays always appears on the front of a drawer, i.e., a file can never begin in the middle of a drawer (block). Likewise, an array element cannot overlap from one drawer to the next (all the bytes for an element must exist in the same drawer (block). A specific file name can use as many as 128 drawers (blocks).

The disk directory contains the track and sector (block) associated with each file name. This directory is consulted by the operating system (FDOS) each time a file name is referenced by the program. File names are associated with a channel number (file no.). To make our analogy complete, let us further imagine that the file room (containing the 80 cabinets) is located in a very large building and you (the program) are located away from the storage room and it is actually possible for you to take any one of sixteen different routes (channels 1 through 16) to reach the storage room. Before you store data or retrieve data from a file drawer (name) it is necessary to specify which route (channel no.) will be used to transport the data. As long as a file name has a drawer OPEN, the designated route (channel no.) cannot be used for another file name until the channel is CLOSEed, and the next file OPENed using the same channel number.

Finally, imagine that you have a utility cart (buffer) to transport one file drawer (512 byte block) back and forth between you (main memory), and the disk (storeroom).

In actuality, the contents are never removed from a disk block. Instead, its contents are copied onto the main memory buffer and when the buffer is sent back to update the disk, the buffer contents are copied back into the file on the disk. The temporary file is the same size as the entire permanent file.

Virtual Array File Organization

When a virtual array file is opened, BASIC creates a 512-byte (1 block) buffer in main memory to hold the block of the file currently being used (one buffer is created for each virtual array file). Each file is considered to consist of a sequence of bytes, numbered 0 (first block) to n (last block). The description of each virtual array contains the channel number to which the file containing the array is attached, and the address within the file at which the array starts (the array's base address).

The base address for a virtual array is determined when the DIM statement declaring the array is processed. The base address assigned is the next available (higher) address which will not cause an array element to cross a block boundary. Each array element must be wholly contained within a 512-byte block. This restriction may be defined as: the base address of an array must be an integral multiple of the array element length and no array element may be longer than 512 bytes. This works since all virtual array elements have a length which is an integral power of 2.

Since virtual arrays are assigned addresses in the file in the order in which the arrays are declared in the DIM statement, the restrictions noted in the paragraph above suggest that it is possible to allocate file space efficiently or inefficiently when arrays having differing element lengths are assigned to the same file. This depends on the order in which array declarations appear in the DIM statement. To eliminate wasted file space, the simplest rule is that virtual array declarations should appear in the DIM statement (as read from left to right) in decreasing order of array element lengths. This rule ensures that if an element overlaps a block boundary, a minimum of space is left unused in the previous block.

NOTE

The unused space at the end of a virtual array file is available for use if a subsequent DIM statement enlarges the file. See the discussion that follows on equivalent virtual arrays.

In the following DIM statement, the arrays are allocated space as shown below the statement.

```
DIM #1%, A$ (10%) = 64%, B (10%, 9%), C% (1%, 4%)
```

A$	11 elements of 64 bytes each	= 704 bytes
B	110 elements of 8 bytes each	= 880 bytes
C%	10 elements of 2 bytes each	= 20 bytes
	TOTAL:	1604 bytes

The total space needed is 68 bytes greater than 3 blocks (1604 – (3 * 512)). The blocks will be allocated as follows. Note that the first three blocks are completely used, leaving only the extra 68 bytes for the fourth block. The 444 bytes remaining in the fourth block are unused.

BLOCK	VARIABLE	ELEMENTS	BYTES
1	A$	8	512
2	A$	3	192
2	B	40	320
3	B	64	512
4	B	6	48
4	C%	10	20

Suppose the DIM statement is changed to read:

```
DIM #1%, C% (1%, 4%), B (10%, 9%), A$ (10%) = 64%
```

The total space needed remains 1604 bytes. The variables however, are allocated to blocks as follows.

BLOCK	VARIABLE	ELEMENTS	BYTES
1	C%	10	20
1	unused	—	4
1	B	61	488
2	B	49	392
2	unused	—	56
2	A$	1	64
3	A$	8	512
4	A$	2	128

Only 508 bytes of block 1 and 456 bytes of block 2 are used. The unused portions, totalling 60 bytes, could not entirely contain one more data element in the sequence assigned. As a result, block 4 has only 384 bytes available, instead of the possible 444.

Storage Size Requirements

Before a virtual array can be dimensioned, space must be reserved for it on the disk. When a new virtual array file is OPENed, the largest available contiguous space on the file-structured storage medium is allocated for the single file, unless the SIZE is included in the OPEN statement. If the NEW files are OPENed without a SIZE statement and there is only one contiguous space available on the device, BASIC will display "? I/O error 306..." telling you there is no more room on the storage device, when the attempt is made to OPEN the second file. This will happen even though there is enough room on the disk for all the data you plan to store in each of the files.

Virtual Array SIZE Calculation

SIZE must be stated as an integer number of BLOCKS (1 BLOCK = 512 BYTES). A virtual array file may contain more than one array. file SIZE must be large enough to equal or exceed the total number of storage bytes required by all of the floating-point (real) elements, integer elements and string elements you plan to use in the virtual array file. Each floating-point element occupies 8 bytes. each integer element occupies 2 bytes. Each string element occupies 1 byte/character. Elements are not allowed to overlap block boundaries. If an element is too large to fit in the remaining storage space in a block, the remaining space is left vacant and the element is placed in the next higher block. To elminiate wasted file space, the simplest rule is that virtual array declarations should appear in the DIM statement from left to right in decreasing order of array element lengths.

Appendix J lists a program which allocates virtual arrays by blocks according to your DIM statement.

String elements require special consideration since all elements for a given string array variable name must be the same number of characters in length *and* that length must be 2 or 4 or 8 or 16 or 32 or 64 or 128 or 256 or 512 characters, i.e., any power of 2 between w and 512 inclusive. String element length is assigned in the DIM statement. If no length is specified, the default length is 16 characters.

Based on the above considerations, and given:

R = the No. of REAL elements in all FLOATING-POINT arrays
I = the No. of INTEGER elements in all INTEGER arrays
S = the No. of STRING characters in all STRING arrays
V = vacant bytes prior to boundaries of occupied blocks

the formula becomes:

SIZE = (R * 8 + I * 2 + S + V) BYTES/512 BYTES/BLOCK

Example

```
DIM #1, A(10), AA(5,6), B%(30), BB%(3,7), A$(10) = 8, AA$(50,50) = 2
```

Calculate the SIZE for the following virtual array dimension statement:

R = 11 + (6 * 7) = 53 elements for A(10) and AA(5,6)
I = 31 + (4 * 8) = 63 elements for B%(30) and BB%(3,7)
S = 11 * 8 + (51 * 51 * 2) = 5290 characters for A$(10) and AA$(50,50)

V = 0

Block Allocation Table

424 Bytes	88 Bytes	38 Bytes	80 Bytes	394 Bytes	4608 Bytes	288 Bytes	224 Bytes
53 Reals	44 Integers	19 Integers	10 8 Char. Strings	197 2 Char. Strings	2304 2 Char. Strings	144 2 Char. Strings	224 Vacant Bytes
Block 1		Block 2			Blocks 3 thru 11	Block 12	

NOTE
Remember element numbering starts with 0, e.g., DIM #1, A(10) yields 11 elements; DIM #1, A(0) yields one element.

SIZE = (53 * 8 + 63 * 2 + 5290) / 512 = 11.40625 BLOCKS

Only whole blocks may be allocated, i.e., SIZE must be an integer, the next higher whole number is the correct answer: SIZE = 12 BLOCKS.

CREATING VIRTUAL ARRAYS

Creating virtual arrays requires the following actions:

1. A filename must be associated with the virtual arrays. (OPEN {filename} AS... statement.)

2. A channel number must be associated with the filename. (FILE n clause of OPEN... statement.)

3. A determination must be made to use NEW data (create a new disk file for new data) or OLD data (data already in a virtual array disk file). (AS {NEW, OLD} clause of OPEN statement.)

4. The SIZE of the arrays in blocks should be stated. (SIZE clause of OPEN statement.)

5. The arrays must be DIMensioned. (DIM statement)

6. The DIMension must be associated with the channel number picked in step 2, above. (#n clause of DIM statement.)

7. The channel (file) must be CLOSEd in order to transport the most current array data (contained in the buffer) to the disk.

The following example program illustrates the process of creating a virtual array.

```
10  CLOSE 1          ! insurance
15  ! new file, "test.vrt", virtual array, chan 1, size = 1 block
20  OPEN "TEST.VRT" AB NEW DIM FILE 1 SIZE 1
30  DIM #1, A(5%)          ! dimension array a, 5 elements, chan 1
40  FOR I% = 0 TO 4%       ! loop to load array
50  A(I%) = I%             ! assign value
60  NEXT I                 ! loop
70  PRINT A(0..4)          ! print array
80  CLOSE 1                ! close channel/file
90  END
```

USING VIRTUAL ARRAYS

Once a virtual array file has been opened and a DIM statement for the channel has been executed, the virtual array elements may be used just as ordinary variables.

Using Virtual Arrays as Ordinary Variables

In the following example, elements of A% may be used wherever integer array elements may be used, except in RBYTE, WBYTE, or CALL statements. (See the section on IEEE-488 Bus Input and Output Statements.)

```
10      OPEN "INTEGR.BIN" AS DIM FILE 1%
20      DIM #1%, A%(255%, 127%)
```

The following example shows that data may be read from or written to the file simply by writing the array name in an expression or assigning a value to an array element.

```
305     IF A%(I%, J%) ) 0% THEN 350
430     LET A%(K%, 0%) = ABS (A% (K%, 1%)) + 2%
```

Using Virtual Array Strings

Strings in virtual arrays are considered by BASIC to be of fixed length. The default length is 16 characters, or as declared in the DIM statement (see the DIM discussion in this section).

The following example specifies a virtual string array with character elements. Line 390 will display the number 32 regardless of the value assigned to that particular element of A$.

```
380    DIM #2%, A$(15%) = 32%
390    PRINT LEN(A$(I%))
```

When characters are assigned to a virtual array string element, BASIC will add null characters to the right end of the string until it equals the declared string element length. This can be the source of subtle errors. Consider the virtual array A$ of the previous example. The following program section attempts to add an * character to all of the elements of A$. This example will not work, and results in error 904 (string too long for virtual array string field).

```
570    FOR I% = 0% TO 15%
580    A$(I%) = A$(I%) + "*"
590    NEXT I%
```

Each element of A$ is allocated 32 characters. The expression A$(I%) + "*" results in a 33-character string. When this string is assigned to A$(I%), error 904 (string too long for virtual array string field) results. It is necessary to strip trailing null bytes from the virtual array string value before appending the '*' character.

The TRIM statement causes trailing null bytes to be removed from a virtual array string. The format of the TRIM statement is:

```
TRIM t%
```

where t% specifies whether or not to trim trailing null bytes from a virtual array string. If t% is zero, no trimming is performed. If t% is not zero then all trailing null bytes are trimmed.

The TRIM statement must be issued before reading the first virtual array string element for which null byte trimming is desired.

PROGRAMMING TECHNIQUES

The key to the efficient use of virtual arrays is to minimize the number of data transfers to or from the virtual array file. Whenever a virtual array element is accessed, either to read its value or to assign to it a new value, BASIC determines the block number and the file in which the element exists. If the block required is not in the memory buffer, the required block is moved from the file into the memory buffer. The block previously held in the buffer is written to the file only if a change in its contents occurred.

Array Element Access

Array elements in a file are stored in row-major order which means that access to the elements, in the storage order, is most efficient when the rightmost array subscript varies most quickly, as in the following example:

```
4650   FOR I% = 0% TO 63%
4660     FOR J% = 0% TO 63%
4670       A% (I%, J%) = 0%
4680     NEXT J%
4690   NEXT I%
```

The form just discussed describes the most efficient access method for initializing the array A%. When the array A% is stored on a floppy disk, its initialization required 7.05 seconds. When the same array is stored on the E-disk, its initialization required 4.71 seconds. The following example is the least efficient access method:

```
4650   FOR I% = 0% TO 63%
4660   FOR J% = 0% TO 63%
4670   A% (J%, I%) = 0%
4680   NEXT J%
4690   NEXT I%
```

This second example requires 254.33 seconds (8.05 seconds on ED0:), 36 times as long (1.7 times for ED0:) as the first example. The first example requires a new block to be read for every 256 elements of A% that are written. This second example, however, requires that a new block be read for every four elements of A% that are written.

Splitting Arrays Among Files

If information stored in different virtual arrays will often be required at the same time, placing the arrays in separate files will speed processing. The following example illustrates the value of utilizing separate files for two parallel virtual arrays as opposed to placing both arrays within the same file.

Example

In some programs it may not be possible to use the null stripping function described previously in this section. This may be because the null characters are part of the string data required. However, the true string length can be determined without stripping the null characters. If there is no character which may be used to specify the end of a string in the virtual array, this information may be retained by placing an integer array parallel to the string array. Thus, for each string element, an integer element contains the length of the string.

In the following example, L%(I%) contains the length of string A$(I%). The significant characters of element A$(I%) can be recovered by a statement such as line 6370.

```
6365   OPEN "DATA.VRT" AS DIM FILE 1%
6360   DIM #1%, A$(1023%) = 16%, L%(1023%)
6370   B$ = LEFT (A$(I%), L%(I%))
```

A drawback to this method is that each time an element of A$ is read from the beginning of the file, another block of the file must be read to retrieve the corresponding element of L%.

A more efficient organization is to assign A$ to one virtual array file and L% to another file. This will cause the BASIC Interpreter to assign two buffers, one for the strings of A$ and one for the integers of L%.

```
4785   OPEN "DATA1.VRT" AS DIM FILE 1%
4790   OPEN "DATA2.VRT" AS DIM FILE 2%
4770   DIM #1%, A$(1023%) = 16%
4780   DIM #2%, L%(1023%)
```

In an actual test, accessing the elements of A$, one by one in increasing order, the first example (with A$ and L% in one file) required 386 seconds. With A$ and L% in separate files, only 15.8 seconds were required to perform that same processing, a 24:1 difference.

Reusing Virtual Array Declarations

When a virtual array DIM statement has been executed, the variables defined as virtual arrays remain defined even after the virtual array file has been closed. However, an attempt to access a virtual array when the channel has been closed will result in an error 308 (channel not open). Since the variables remain defined, it is possible to close a virtual array file and to later re-open it. Since the variables have already been defined, a new DIM statement is not necessary to re-open the file. Note, however, that the file must be re-opened using the same channel number as that used in the DIM statement defining the arrays.

If a number of separate virtual array files, all with the same data organization, must be processed by a program, it is possible (if no two files are to be processed simultaneously) to reuse the variable definitions. Consider the following processing sequence:

```
OPEN      first file
DIM       virtual arrays
          process first file
CLOSE     first file
OPEN      second file (on same channel as 1st file)
          process 2nd file
CLOSE     second file
  .
  .
  .
```

In each processing loop, the same virtual array variables may be used to process the file data.

Equivalent Virtual Arrays

It is possible to execute multiple DIM statements for a single channel. The second (and subsequent) DIM statements for a channel simply redefine the file organization as shown below. The first DIM statement allocates a 441-element array, A%, organized as a 21x21 matrix. The second DIM statement does not allocate array B% following A% but redefines the 441 integer elements of the file as a vector with the name B% (similar to a FORTRAN equivalence).

```
110     DIM #1%, A% (20%, 20%)
120     DIM #1%, B% (440%)
```

NOTE

Fluke BASIC makes no check for the consistency of virtual array equivalences. This is the responsibility of the programmer.

Section 8
Functions

CONTENTS

Introduction 8-3
Overview 8-3
General Purpose Mathematical Functions 8-4
ABS() Function 8-5
ASH() Function 8-5
EXP() Function 8-5
INT() Function 8-5
LN() Function 8-6
LOG() Function 8-6
LSH() Function 8-6
MOD() Function 8-6
RND Function 8-7
SGN() Function 8-7
SQR() Function 8-7
Trigonometric Functions 8-8
ATN() Function 8-9
COS() Function 8-9
SIN() Function 8-9
TAN() Function 8-9
String Functions 8-10
ASCII() Function 8-11
CHR$() Function 8-11
CPOS() Function 8-11
DUPL$() Function 8-11
INSTR() Function 8-12
LCASE$() Function 8-12
LEFT() Function 8-12

LEN() Function 8-12
MID() Function 8-12
NUM$() Function 8-13
RAD$() Function 8-13
RIGHT() Function 8-13
SPACE$() Function 8-13
TAB() Function 8-13
UCASE$() Function 8-14
VAL() Function 8-14
User-Defined Functions 8-14

INTRODUCTION

Functions are predefined operations available in Fluke BASIC by applying a function name to an appropriate set of arguments. The availability of a wide range of functions can significantly simplify a programming task. Mathematical functions perform calculations on numeric quantities. String functions create, manipulate, measure, and extract portions of character strings. In addition, Fluke BASIC also allows the user to define specialized functions to meet the needs of particular data processing and instrumentation control tasks. Refer to the discussion of the DEF FN statement at the end of this section and in the Reference volume of this manual set.

OVERVIEW

This section is divided into four subject areas: General Purpose Mathematical Functions, Trigonometric Functions, String Functions and User-defined Functions. Effective use of functions requires a clear understanding of proper data types and formats for input and expected results.

GENERAL PURPOSE MATHEMATICAL FUNCTIONS

The general purpose mathematical functions supplied with Fluke BASIC include an assortment of tools to simplify computation tasks on collected data. Included are a square root function, natural and common logarithms, exponentiation of e, absolute value, sign, and greatest integer.

Mathematical functions perform calculations on numeric quantities.

- Mathematical functions operate on integer or floating-point numbers.
- Mathematical functions accept as input an expression that evaluates to a numeric quantity.
- Mathematical functions return an integer or floating-point number.
- Conversion between numeric data types (i.e., floating-point to integer) is automatic where required by the function or specified by the user. This conversion process requires additional processing time.
- The domain of acceptable input values for some functions is limited or not continuous.
- The range of resulting output values for some functions is not continuous or has points of underflow (too close to zero) or overflow (too large).
- The definition of each function includes limitations of domain and range.

Following is a summary of the Fluke BASIC General Purpose Mathematical Functions. Each function is described briefly and its format is shown. All functions are fully described in the BASIC Reference manual.

ABS() Function

Format: ABS(numeric expression)

The ABS function returns the absolute value of a floating-point or integer number. ABS has a floating-point number or integer, as an argument, and returns a positive floating-point number or integer as a result.

ASH() Function

Format: ASH(x%,count%)

The ASH function operates on binary integers (x%) by arithmetically (signed) shifting them by count% bits. The ASH function arguments are integers and any floating-point arguments are truncated to integer values.

When count% is positive, a left shift is performed count% places and zeroes are shifted in from the right. When count% is negative, a right shift is performed by the number of places specified and the sign bit is shifted in from the left. When count% is zero, no shift is performed and x% is returned.

EXP() Function

Format: EXP(numeric expression)

The EXP function returns a floating-point number equal to the result of raising the number e to an exponential power equal to the given input value. EXP has a floating-point number as an argument, and returns a floating-point number.

INT() Function

Format: INT(numeric expression)

The INT function returns the greatest integer less than or equal to a given floating-point number. If INT is given a floating-point argument, a floating-point result is returned. An integer argument is returned unchanged, as an integer.

LN() Function

Format: LN(numeric expression)

The LN function returns a floating-point number equal to the natural logarithm (base e) of the input value. LN has a floating-point number as an argument, and returns a floating-point number.

LOG() Function

Format: LOG(numeric expression)

The LOG function returns a floating-point number equal to the common logarithm (base 10) of the input value. LOG has a floating-point number as an argument, and returns a floating-point number.

LSH() Function

Format: LSH(x%,count%)

The LSH function operates on binary integers (x%) by logically (unsigned) shifting them by count% bits. The LSH function arguments are integers and any floating-point arguments are truncated to integer values.

When count% is positive, a left shift is performed count% places and zeroes are shifted in from the right. When count% is negative, a right shift is performed by the number of places specified and zeroes are shifted in from the left. When count% is zero, no shift is performed and x% is returned.

MOD() Function

Format: MOD(x,y)

The MOD function returns the remainder produced by dividing two integer or floating-point numbers (x,y). The MOD function is defined as:

$$\text{mod}(x,y) = x - y * \text{truncate}(x/y)$$

in which the truncate operation drops any fractional result produced by the division of x by y. The result always has the same sign as x. An integer result is returned if x and y are both integers. A floating-point result is returned if either x or y is floating-point.

RND Function

Format: RND

The RND function uses an internal algorithm to produce a pseudo-random number that is greater than zero and less than one. RND is not a true function since it does not operate upon an argument. The sequence of values returned by RND is repeatable unless by a RANDOMIZE statement is used in the program each time the program is run.

SGN() Function

Format: SGN(numeric expression)

The SGN function returns the sign of a floating-point or integer number. This is called the signum function. SGN has a floating-point or integer number as an argument, and returns one of three integers: 1 if the argument was positive, 0 if the argument was zero, or –1 if the argument was negative.

SQR() Function

Format: SQR(numeric expression)

The square root function returns a floating-point number equal to the square root of the input value. SQR has a floating-point number as an argument, and returns a floating-point number.

TRIGONOMETRIC FUNCTIONS

Fluke BASIC also includes four trigonometric functions: sine, cosine, tangent, and arctangent. These functions have some discontinuities and limits that can produce unexpected errors when used improperly. The discussions that follow define these factors.

Trigonometric functions in Fluke BASIC use radian measure for angular quantities.

- The value PI stored by Fluke BASIC (3.14159265358979) represents an approximation of the ratio between the circumference and the diameter of any circle.

- Radian measure defines the angular distance around a circle as 2*PI radians. This is the equivalent of 360 degrees.

- Angles are thus easily defined as fractional parts of PI. For example:

  PI radians = 180 degrees
  PI / 2 radians = 90 degrees

- To convert from degrees to radians, multiply by (PI / 180).

- To convert from radians to degrees, multiply by (180 / PI).

- The descriptions of trigonometric functions that follow use radian measure exclusively.

Following is a summary of available Trigonometric Functions found in Fluke BASIC. A complete description of each function may be found in the BASIC Reference manual.

ATN() Function

Format: ATN(numeric expression)

The ATN function returns the principal arctangent (inverse tangent) in radians of a floating-point numeric input value. ATN has a floating-point number as an argument, and returns a floating-point number.

COS() Function

Format: COS(numeric expression)

The COS function returns the cosine of an angle that is expressed in radians. COS has a floating-point number as an argument, and returns a floating-point number.

SIN() Function

Format: SIN(numeric expression)

The SIN function returns the sine of an angle that is expressed in radians. SIN has a floating-point number as an argument, and returns a floating-point number.

TAN() Function

Format: TAN(numeric expression)

The TAN function returns the tangent of an angle that is expressed in radians. TAN has a floating-point number as an arguments, and returns a floating-point number.

STRING FUNCTIONS

String functions create, manipulate, and extract portions of character strings.

- String functions operate only upon strings, but can have floating-point numbers, integers, or character strings in their argument list.
- Integer results from string input include:

  ASCII (decimal) value of a character.

  Length of a string.

  Number of characters within a string at which a substring was located.
- String results from integer input include:

  Character corresponding to an ASCII (decimal) value.

  Specified left, right, or center substrings.

  Space character strings, of either specified length or to a specified column position.

  Escape code sequence for positioning the display cursor.
- A string result is available in the format that PRINT or PRINT USING would display a number.
- Floating-point numbers may be used in place of integer input. BASIC will truncate as necessary and convert to integer form (requires additional processing time). Note that truncation to integer form may cause unexpected results.
- Numeric expressions may be used in place of floating-point or integer numeric input.
- See Appendix G, ASCII/IEEE-488 Bus Codes, for a chart of ASCII characters and corresponding decimal numbers.

Following is a synopsis of the String Functions included in Fluke BASIC. Each function is described fully in the BASIC Reference manual.

ASCII() Function

Format: ASCII(A$)

The ASCII function returns an integer equal to the ASCII (decimal) value of the first character in the specified string. ASCII has a string as an argument and returns an integer result. Refer to Appendix G, ASCII/IEEE-488 Bus Codes, for a chart of ASCII characters and corresponding decimal numbers.

CHR$() Function

Format: CHR$(x%)

The CHR$ function returns an ASCII string character corresponding to an integer input. CHR$ has an integer as an argument, and returns a single-character string (8-bit binary pattern). The CHR$ function is the logical opposite of the ASCII function.

NOTE

Some integers used in the CHR$ function result in display commands when printed and do not appear on the screen. For example PRINT CHR$(7) activates the beeper.

CPOS() Function

Format: CPOS(R%,C%)

The CPOS function returns a string which directly positions the display cursor when printed. CPOS has two integers as arguments (R% = row, C% = column)and returns an 8-character string.

DUPL$() Function

Format: DUPL$(A$,n%)
DUPL$(val%,
n%)

The DUPL$ function returns a string of n% duplicated characters or strings (A$). When the first argument to DUPL$ is val%, val% is first converted to its ASCII character equivalent (as in the CHR$() function). Thus, DUPL$("a",5%) returns "aaaaa".

INSTR() Function

Format: INSTR (x%,A$,B$)

The INSTR function searches for a specified substring (B$) within a string (A$) and returns the starting location of the substring. The substring search begins at the character position specified by x%. INSTR returns an integer result.

LCASE$() Function

Format: LCASE$(A$)

The LCASE$ function converts its alphabetic string argument (A$) to lower-case. The range of ASCII characters affected is A to Z.

LEFT() Function

Format: LEFT(A$,n%)

The LEFT function returns a substring of the specified string (A$), starting from the left. The number of characters returned is, if possible, the number specified by the second argument (n%). LEFT has a string and an integer as arguments, and returns a string result.

LEN() Function

Format: LEN(A$)

The LEN function returns the number of characters contained in a string (A$). LEN has a string as an argument, and returns an integer. The string count includes leading and trailing blanks and null characters.

MID() Function

Format: MID(A$,n1%,n2%)

The MID function returns a substring of the specified string (A$), starting from the specified character position n1%), and including the specified number of characters (n2%). MID has as arguments a string and two integers, and returns a string.

NUM$() Function

Two
Formats: NUM$(x% or x)
NUM$(x% or x,A$)

The NUM$ function returns a string of characters in the format that a PRINT or PRINT USING statement would output the given number. NUM$ has as arguments an integer (x%) or a floating-point number (x), and an optional character format string (A$), and returns a numeric character string with appropriate spaces and decimal.

RAD$() Function

Format: RAD$(integer%,base%)

The RAD$ function returns a string that corresponds to integer% using base%. Digits greater than 9 in the output string are returned as "A" to "Z" floating-point arguments are first truncated to integer values.

RIGHT() Function

Format: RIGHT(A$,n%)

The RIGHT function returns a substring of the specified string (A$), starting from the specified character position (n%), to the right end of the string. RIGHT has as arguments a string and an integer, and returns a string.

SPACE$() Function

Format: SPACE$(x%)

The SPACE$ function returns a string of spaces as specified. SPACE$ has as argument an integer, and returns a string of spaces.

TAB() Function

Format: TAB(x%)

The TAB function returns a string of spaces that would advance the current print position on an external printer to one column past the specified column number. TAB has an integer as an argument, and returns a string of spaces that may be preceded by Carriage Return and Line Feed.

UCASE$() Function

Format: UCASE$(A$)

The UCASE$ function converts its alphabetic string argument (A$) to upper-case. The range of ASCII characters affected is a to z.

VAL() Function

Format: VAL(A$)

The VAL function returns the floating-point numeric value of a numeric string (A$). VAL has a string as an argument and returns a floating-point number.

USER-DEFINED FUNCTIONS

Format: DEF FN{variablename}(argument[s]) {function definition}

The DEF FN statement allows functions to be defined for subsequent use in expressions. The function name can be any variable name, such as A%, FA$, Z, including variables previously used in the program. (e.g., FNA% does not conflict with A% already existing in the program.)

The DEF FN statement applies the function defined by the function definition to the argument[s] supplied. The arguments may be any of the BASIC variable types and the result may also be any of the BASIC variable types.

Section 9
IEEE-488 Bus I/O

CONTENTS

Introduction 9-3
Overview 9-3
IEEE-488 Addressing 9-4
  Device Addressing 9-5
  Addressing Serial IEEE-488 Devices 9-7
  Port Addressing 9-8
Initialization and Control Statements 9-9
  CLEAR Statement 9-10
  INIT Statement 9-10
  LOCAL Statement 9-11
  LOCKOUT Statement 9-11
  ON PORT Statement 9-12
  OFF PORT Statement 9-12
  PASSCONTROL Statement 9-12
  REMOTE Statement 9-12
  SET SRQ Statement 9-13
  TRIG Statement 9-13
  TERM Statement 9-13
  TIMEOUT Statement 9-13
Input and Output Statements 9-14
  INPUT Statement 9-14
  INPUT LINE Statement 9-15
  INPUT WBYTE Statement 9-15
  INPUT LINE WBYTE Statement 9-15
  PRINT and PRINT USING Statements 9-16
IEEE-488 Data Transfer Statements 9-17
  RBYTE Statement 9-18

WBYTE Statement or Clause 9-18
RBYTE WBYTE Statement 9-18
RBIN Statement 9-19
RBIN WBYTE Statement 9-19
WBIN Statement 9-20
IEEE-488 Polling Statements 9-20
CONFIG Statement 9-21
ON SRQ and OFF SRQ Statements 9-21
ON PPOL and OFF PPOL Statements 9-21
PORTSTATUS() Function 9-22
PPL() Function 9-22
SPL Function 9-22
WAIT Statement 9-23

INTRODUCTION

The IEEE-488 bus standard is the backbone of a programmable instrumentation system. Fluke BASIC has many statements intended specifically for controlling instruments connected to the IEEE-488 bus.

Detailed information on instrument bus communication concepts and messages, control lines and data lines, and on timing, is available in the standard "IEEE-488-1980 Standard Digital Interface for Programmable Instrumentation". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York 10017. Appendix B gives the correspondence between BASIC commands and the sequence of bus actions which actually take place. Other material may be found in the 1722A System Guide, Fluke Application Bulletin AB-36, Fluke Technical Bulletin C0076 and Appendices B through D of this manual.

OVERVIEW

Communications on the bus occur on a detailed signal and code level as defined by the standard. However, for the most part, bus communication can be viewed on a functional level. On this level the user is concerned more about what is happening on the bus than the actual processes. Bus communications are categorized as functional messages exchanged between devices. Each bus device in a system may be designed to implement only the messages which are important to its system purpose.

This section describes the statements which are designed for communication with instruments on the IEEE-488 buses. These statements allow the Instrument Controller to control any type of bus-compatible device. The following discussion of IEEE-488 bus commnuications centers first on device and port addressing then on the BASIC statements used for IEEE-488 bus communications. The BASIC statements used for IEEE-488 bus communications are divided into four groups: Initialization and Control, Input and Output, IEEE-488 Data Transfer, and IEEE-488 Polling.

IEEE-488 ADDRESSING

The following paragraphs describe in general terms how BASIC communicates directly with the IEEE-488 bus port or specific IEEE-488 devices. The general discussion is followed by a descriptions of device addressing, serial device addressing, and port addressing. Device and port addresses are an integral part of many of the IEEE-488 related statements.

- The Instrument Controller has one IEEE-488 instrumentation bus interface (port). A second IEEE-488 bus port may be added as an option.
- Instruments connected to the controller via the IEEE-488 bus may be addressed using their device address.
- After an instrument or group of instruments has been addressed, the port address may be used for communications without device addressing.
- Serial instruments, such as printers, may also be connected to the controller via the IEEE-488 ports.

The INPUT, PRINT, RBIN, and WBIN statements are used to transfer data to and from the Instrument Controller. Unlike the TRIG or CLEAR statements, for example, these statements are not necessarily tied to the IEEE-488 bus Controller in Charge (CIC) functions. The @ device address specifier is used with these statements to indicate that both of the following functions are to be performed:

1. Address one or more instruments as talkers or listeners. This is a CIC function.
2. Transfer data to or from the IEEE-488 bus. This is a device function.

The INPUT, PRINT, RBIN, and WBIN statements may also be used without device addressing. This is not a CIC function, thus it may be performed when the Instrument Controller is not the CIC, as is the case after the PASSCONTROL statement is issued.

- The PORT statement modifier is used to supress device addressing.
- This addresses the last instrument or instruments that were addressed on the port specified by the PORT statement modifier.

Device Addressing

Instruments (devices) connected to the bus are addressed using their primary and/or secondary address. The following paragraphs describe this process.

- Each device on a bus connected to a port has a primary address in the range of 0 through 30. The address is normally determined by switch settings on or in the instrument. A BASIC command indicates a primary device address with the @ device address specifier:

```
@ {device number}
```

- The device number must correspond with the address switch settings in the instrument.

- The hundreds digit of the device number determines the port assignment of the instrument being addressed: 0 for PORT 0, 1 for PORT 1 (default is 0).

- A device number is any numeric expression with a value in the range 0 through 30 for an instrument connected to PORT 0, or in the range 100 through 130 for an instrument connected to PORT 1. The last two digits are the device number.

- The IEEE-488 bus standard also allows instruments to have one or more secondary addresses. The function associated with a secondary address depends on the instrument design. A BASIC command identifies a secondary address as part of the device number:

```
@ {device number: secondary number}
```

- The secondary number is any numeric expression with a value in the range 0 through 31. The syntax is:

```
@ {device: secondary: secondary}
```

- More than one device may be addressed at once by listing device numbers in the command. The @ character is the only separator in a multiple device list (commas are not used). Secondary address numbers are optional. The device list syntax is as follows:

```
@ primary address: secondary addr1: secondary addr2: ...
```

  For example, the statement:

```
PRINT @1:3 @104, m$
```

  sends the string m$ to device 1, secondary address 3 on port 0 and to device 4 on port 1.

- A device attached via the IEEE-488 bus may also be attached as a serial device.

Addressing Serial IEEE-488 Devices

Devices connected to the IEEE-488 bus as serial devices use a slightly different addressing form. This address form associates the IEEE-488 device with a channel number for serial I/O.

- The channel must have been previously opened with the OPEN statement. Use the # statement modifier when using this form of addressing (for example, PRINT #). For example:

```
500 OPEN "GP0:3" AS NEW FILE 10%
510 PRINT #10%, "MESSAGE 1"
```

- The "file name" of an IEEE-488 device provides both the port number and device address of the instrument. The file name is:

  GPn:p.s

  where:

  n is the IEEE-488 "port number", which ranges from 0 to 9.

  p is the "primary device address", which ranges from 0 to 30.

  s is the "secondary device address", which ranges from 0 to 31.

- A primary address must be specified if a secondary address is used. Thus, the address "GP0:.3" is illegal.

- All device addressing may be suppressed by using a "file name" of "GPn:", which simply reads from or writes to a IEEE-488 port without doing any addressing.

- Secondary addressing is suppressed by omitting the ".s" portion of the IEEE-488 "file name".

- When the "@" form of device addressing is used the PRINT statement uses the "," format specifier to indicate that the EOI (End or Identify) message is to be sent with the last character of a print string. This will not occur when using the "#" form of addressing. This ensures that a PRINT statement directed to a printer will operate in the same way whether, say, a printer is interfaced via the RS-232 port or via the IEEE-488 bus.

Port Addressing

Port addressing is used to communicate with an instrument or group of instruments after they have been previously addressed, without re-addressing them. This form of addressing suppresses individual device addresses and communicates directly with the named port.

The IEEE-488 port(s) are addressed by BASIC with the PORT statement modifier (port expression). A port expression takes the form:

```
PORT {numeric expression}
```

- The value of the numeric expression used in a statement must be either 0 or 1. Refer to the 17XX Instrument Controller System Guide for identification of bus ports and information on cable connections.

- Once a device has been addressed, the PORT statement modifier may be used to communicate directly with the device, bypassing any device addressing. For example, the statement:

  ```
  PRINT @4, A$
  ```

  prints string A$ to device number 4 on IEEE-488 port 0. Now the same device may be addressed in a subsequent statement with:

  ```
  PRINT PORT 0, A$
  ```

  This form of the PRINT statement sends the string A$ to port 0 without performing any device addressing.

INITIALIZATION AND CONTROL STATEMENTS

The following paragraphs summarize Fluke BASIC statements which initialize, set up, or otherwise manipulate instruments either before or after data transfers. These statements are also listed in Table 9-1. Complete descriptions of each statement may be found in the Reference Section of this manual.

Table 9-1. Initialization and Control Statements Summary

STATEMENT	DESCRIPTION
CLEAR	Sends Device Clear or Selected Device Clear bus messages. Sets instruments to a ready state.
INIT	Halts port activity and prepares port for further messages.
LOCAL	Sets Remote Enable bus line false or sends Go To Local bus message to instruments in device list.
LOCKOUT	Disables local switch on all addressed instruments.
PASSCONTROL	Designates another IEEE-488 bus instrument to act as Controller in Charge of the interface.
REMOTE	Sets Remote Enable bus line true. Addresses any devices specified as listeners.
SET SRQ	Requests service from the current Controller in Charge of the IEEE-488 bus interface.
TERM	Selects a terminator character for designating the end of an input stream from a device.
TIMEOUT	Sets a limit on the time the Instrument Controller will wait for a response to a request.
TRIG	Addresses instruments in the device list and sends Group Execute Trigger bus message.

CLEAR Statement

The CLEAR statement sends a device clear or selected device clear message to a specified port or device. The two forms of the CLEAR statement are described below.

Usage: CLEAR [PORT {numeric expression}]

(device clear) message or SDC (selected device clear) message to the specified IEEE-488 port. If the PORT statement modifier is not used, a DCL message is sent to all ports.

Usage: CLEAR {device list}

The CLEAR statement, with a device list, addresses the devices given in {device list} as listeners and sends a SDC (selected device clear) message to them.

INIT Statement

Usage: INIT [PORT {numeric expression}]

INIT (INITialize) sends an IFC (interface clear) message followed by REN (remote enable) and PPU (parallel poll unconfigure) to the specified port or to both ports if not specified.

INIT places the bus in an idle state and sends the following commands to the bus: REN (remote enable), IFC (interface clear), UNL (unlisten), UNT (untalk), and PPU (parallel poll unconfigure).

LOCAL Statement

The LOCAL statement resets instruments to a local state. Typically, this means that front panel controls are activated. Two forms of the LOCAL statement permit port or device addressing.

Usage: LOCAL [PORT {numeric expression}]

This form of the LOCAL statement is the reverse of the REMOTE statement. LOCAL, with a port specified, reverses all effects of the LOCKOUT statement.

When a port is specified, REN is set false on the designated instrument port. If a port number is not specified, REN is set false on both ports.

Usage: LOCAL [device list]

Issuing the LOCAL statement with a device list, sends a GTL (go to local) message to the instruments identified in the device list.

LOCKOUT Statement

Usage: LOCKOUT [PORT {numeric expression}]

LOCKOUT disables not only front panel controls (as with REMOTE) but also any "return to local" function button that may be on an instrument. As defined in the IEEE-488-1978 Standard, any instrument addressed to listen after receiving a local lockout command will immediately be placed in the "Remote With Lockout State". LOCKOUT sets REN, and then sends a LLO (local lockout) message.

This sequence is sent on only one port if a port number is specified or on both ports if the PORT statement modifier is not given.

ON PORT Statement

Usage: ON PORT [p%] GOTO {line number}

The ON PORT statement permits the Instrument Controller to detect commands sent to it by another controller. Port 0 is assumed if p% is not specified. If p% is a floating-point variable, the value is rounded to its integer value. An overflow error may result from this action. Use the PORTSTATUS statement to determine the exact cause of the interrupt.

OFF PORT Statement

Usage: OFF PORT p%

The OFF PORT statement disables a previous ON PORT interrupt. p% must be in the range of 0..1. It is an error for p% to be outside of this range. if p% is omitted, port 0 is assumed.

PASSCONTROL Statement

Usage: PASSCONTROL {device}

The PASSCONTROL statement permits the Instrument Controller to designate another IEEE-488 bus instrument to act as Controller in Charge (CIC) of the interface. The new Controller in Charge is designated in the device clause of the PASSCONTROL statement.

REMOTE Statement

The REMOTE statement sets the REN (remote enable) line on the IEEE-488 bus to true. The two forms of the REMOTE statement are described below.

Usage: REMOTE [PORT {numeric expression}]

When a port number is specified with the REMOTE statement, REN is set true on the specified port. If the PORT statement modifier is not used, REN is set true on both ports.

Usage: REMOTE [device list]

When a device list is specified with the REMOTE statement, REN is set true on the port represented by each instrument specified in the device list. An MLA (my listen address) message is then sent to the listed devices.

SET SRQ Statement

Usage: SET SRQ [PORT {numeric expression}] WITH {status%}

The SET SRQ statement allows the Instrument Controller to request service from the current Controller in Charge (CIC) of the IEEE-488 bus interface. Status% is the "serial poll data byte" returned to the CIC after the CIC performs a serial poll of the Instrument Controller on the specified port. If the PORT statement modifier is not used, port 0 is the default.

TRIG Statement

Usage: TRIG {device list}

TRIG (TRIGger) addresses the set of instruments named in the device list as listeners and then triggers them simultaneously. The effect of the trigger is dependent upon the instrument. For example, a digital multimeter may take a reading, or a source instrument may go from standby to operate.

TERM Statement

Usage: TERM [string]

TERM allows the user to specify an arbitrary 8-bit byte that will also terminate input. The terminating character is Line Feed CHR$(10) when TERM is not used. The EOI line on the bus will always terminate input, regardless of the use of the TERM statement.

TIMEOUT Statement

Usage: TIMEOUT {numeric expression}

TIMEOUT sets a limit on the amount of time the Instrument Controller will wait for a response to an IEEE-488 bus request. This prevents an instrument fault from halting the system. If the TIMEOUT statement is not used, the time allowed defaults to 20 seconds.

INPUT AND OUTPUT STATEMENTS

This discussion summarizes INPUT and PRINT statements as they are used for instruments on the IEEE-488 bus. These statements are described in the Reference Section of this manual. Specific references are provided after each statement definition. Table 9-2 presents a list of the INPUT and PRINT statements. INPUT WBYTE is a specific IEEE-488 bus statement which combines the characteristics of the INPUT statement with a WBYTE clause described later in this section. All of the statements listed in Table 9-2 will accept a port expression as an alternative to a device address. This is described in the Reference Section under "Devices".

Table 9-2. IEEE-488 Bus Input and Output Statements Summary

STATEMENT	DESCRIPTION
INPUT	Accepts data from an instrument.
INPUT LINE	Accepts a full line of data, including trailing carriage return and line feed.
INPUT WBYTE	Outputs a bus message and receives data.
INPUT LINE WBYTE	Outputs a bus message prior to receiving each line of data.
PRINT	Outputs a bus message or data to instruments.
PRINT USING	Outputs data in the specified format to instruments.

INPUT Statement

Usage: INPUT {device}, {input variable list}
INPUT [PORT {numeric expression}] {input variable list}

INPUT is used to receive data from instruments on the IEEE-488 bus. The only syntax difference in the INPUT statement for IEEE-488 bus instruments is the use of a device specification instead of a channel number.

INPUT LINE Statement

Usage: INPUT LINE {device}, {input variable list}
INPUT LINE [PORT {numeric expression},] {input variable list}

The INPUT LINE construction of the INPUT statement allows binary data to be received from the bus and assigned to a string variable. The only syntax difference in the INPUT LINE statement for IEEE-488 bus instruments is the use of a device specification or port expression instead of a channel number.

INPUT WBYTE Statement

Usage: INPUT WBYTE {device}, {wbyte clause} {input variable list}
INPUT WBYTE [PORT {numeric expression},] {wbyte clause} {input variable list}

The INPUT WBYTE statement transmits a bus message contained in a WBYTE clause prior to receiving each data item. After the WBYTE output, the specified instrument is addressed as a talker and one data item is read. The WBYTE clause is then sent again and the next input data item is read. This process is repeated until the input variable list has been satisfied.

INPUT LINE WBYTE Statement

Usage: INPUT LINE WBYTE {device}, {wbyte clause} {input variable list}

The INPUT LINE WBYTE statement allows binary data to be received from the bus and assigned to a string variable (80 characters maximum length). The INPUT LINE WBYTE statement transmits the bus message contained in the WBYTE clause prior to receiving each data item.

After the WBYTE output, the specified instrument is addressed as a talker and one data line is read. The WBYTE clause is then sent again and the next line of input is read. This process is repeated until the input variable list has been satisfied.

PRINT and PRINT USING Statements

Usage: PRINT {device list}, [print item(s)]

PRINT PORT {numeric expression}, USING {format string,} [print item(s)]

PRINT and PRINT USING are used to output data to designated listeners. The instruments which are to receive output data are specified in a device list or in a port expression.

A PRINT or PRINT USING statement which is followed by a device list addresses the specified devices as listeners. All other devices are commanded to unlisten.

A PRINT PORT or PRINT PORT USING statement communicates with an instrument or group of instruments after they have been previously addressed, without re-addressing them.

Characters are sent exactly as a normal PRINT or PRINT USING statement, except for the use of the comma and semicolon to format the output. A comma following a data item in the output list indicates the EOI bus line is to be set simultaneously with the last character of that item. It does not indicate tabulation to 16 character columns by sending extra spaces.

A Carriage Return and Line Feed, with the EOI bus line set simultaneously with the Line Feed, follows the last data item in a PRINT list when it is not terminated with either a comma or a semicolon.

IEEE-488 DATA TRANSFER STATEMENTS

The following discussion summarizes Fluke BASIC statements which provide direct access to the data lines and some of the control lines of the IEEE-488 Instrumentation bus. These statements allow optimal handling of the binary data sent by some instruments in "high speed" mode. The data transfer statements are summarized in Table 9-3. Two of the statements described below directly handle binary floating-point information as described in the standard "IEEE Floating Point Arithmetic for Microprocessors". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York, 10017. All of the statements listed in Table 9-2 will accept a port expression as an alternative to a device address. This is described earlier in this section.

Table 9-3. Data Transfer Statements Summary

STATEMENT	DESCRIPTION
RBIN	Inputs double and single-precision data in IEEE standard floating point format.
RBIN WBYTE	Outputs data designated by WBYTE prior to receiving double and single-precision data in IEEE standard floating point format.
RBYTE	Inputs binary data bytes from the designated port.
RBYTE WBYTE	Outputs data designated by WBYTE prior to each designated input cycle of binary data bytes.
WBIN	Outputs double and single-precision data in IEEE-standard floating point format.
WBYTE	Sends an integer variable array as bus messages to the designated port.

RBYTE Statement

Usage: RBYTE [PORT {numeric expression},] {integer variable array subrange}

RBYTE (Read BYTE) reads a fixed-length block of arbitrary bit binary data bytes from an instrument. The data from the instrument is placed in a specified one-dimensional integer variable array. The array may not be a virtual array.

WBYTE Statement or Clause

Usage: WBYTE [PORT {numeric expression},] {integer variable array subrange}

WBYTE (Write BYTE) sends an arbitrary set of bus commands or data bytes taken from the specified integer array(s) to a port. The WBYTE statement is also used as a clause within some other IEEE-488 bus control statements. The integer array(s) used may not be virtual arrays.

The ATN, EOI, and data lines of the instrument bus may be set as desired with this command, with the restriction that ATN and EOI may not be set true simultaneously (since this executes a parallel poll).

RBYTE WBYTE Statement

Usage: RBYTE [PORT {numeric expression},] {wbyte clause} {integer variable array subrange}

The RBYTE statement with the added WBYTE clause is used for an instrument that requires an explicit trigger for each reading. The WBYTE clause added to the RBYTE statement provides a means of sending commands or data to a port (via the WBYTE clause) prior to reading the data as specified by RBYTE. The WBYTE data is sent prior to each RBYTE cycle.

RBIN Statement

Usage: RBIN {device}, {variable list:[format specification]}
RBIN @, {variable list:format specification}
RBIN PORT {numeric expression}, {variable list:[format specification]}

The RBIN (Read BINary) statement receives single- and double-precision data in IEEE standard floating-point format from IEEE-488 bus instruments. The specified instrument device number is addressed as a talker. The last port on which IEEE-488 bus I/O was performed is used when the "@" character follows RBIN without a device specified.

A single floating-point value is received from the specified instrument in the format specified in the format specification. The default format is eight-byte double-precision.

RBIN WBYTE Statement

Usage: RBIN {device}, {wbyte clause} {variable list:[format specification]}
RBIN @, {wbyte clause} {variable list:[format specification]}
RBIN PORT {numeric expression}, {variable list:[format specification]}

The RBIN WBYTE statement receives single and double-precision data in IEEE standard floating-point format from IEEE-488 bus instruments. The specified instrument device number is addressed as a talker. The last port on which IEEE-488 bus I/O was performed is used when the "@" character follows RBIN without a device specified. The data specified by the WBYTE clause is sent to the specified port before the RBIN cycle. The WBYTE clause is discussed earlier in this section and in the Reference Section of this manual.

A single floating-point value is received from the specified instrument in the format specified in the format specification. The default format is eight-byte double-precision.

WBIN Statement

Usage: WBIN {device list}, {real variable:[format specification]}
WBIN PORT {numeric expression}, {real variable:[format specification]}

WBIN (Write BINary) sends numeric data to an IEEE-488 bus instrument in single- or double-precision IEEE standard floating-point format. The instrument with the specified device number is addressed as a listener. Instrument addressing is skipped when the @ character follows WBIN without a device specified. Data is then transmitted in the format specified; the default format is eight-byte double-precision.

IEEE-488 POLLING STATEMENTS

The statements summarized in the following paragraphs and in Table 9-4 handle the polling, both serial and parallel, of instruments on the IEEE-488 bus. A detailed description of each statement may be found in the Reference Section.

Table 9-4. IEEE Polling Statements Summary

STATEMENT	DESCRIPTION
CONFIG	Configures an instrument for a parallel poll.
ON PPOL	Enables automatic parallel polls.
OFF PPOL	Disables automatic parallel polls.
ON SRQ	Identifies a service request handling routine and enables the controller to respond to SRQ.
OFF SRQ	Disables service request response.
PORTSTATUS Function	Returns the port status code for the specified port.
PPL Function	Returns parallel poll value.
SPL Function	Returns serial poll value.
WAIT	Halts execution for the specified time or until an enabled interrupt occurs.

CONFIG Statement

Usage: CONFIG {device} {TO numeric expression} {WITH numeric expression}

CONFIG (CONFIGure) either configures or unconfigures an instrument for parallel poll. The TO clause specifies the DIO line on which the instrument should respond to a parallel poll. The WITH clause specifies the active sense (0 or 1) the instrument should use in responding to the poll.

If the "TO line WITH sense" clause is omitted, a PPD (Parallel Poll Disable) message will be sent to the instrument.

ON SRQ and OFF SRQ Statements

Usage: ON SRQ [PORT {numeric expression}] GOTO {line number}
OFF SRQ [PORT {numeric expression}]

ON SRQ (ON Service ReQuest) allows a program to branch to a service request routine when an SRQ (Service Request) is received from an external device. OFF SRQ disables service request interrupt processing. Port 0 is assumed when the port specification is omitted. A resume statement branches back to the interrupted program.

There is no checking for SRQ after the occurrence of the SRQ interrupt and the execution of RESUME.

ON PPOL and OFF PPOL Statements

Usage: ON PPOL [PORT {numeric expression},]GOTO {line number}
OFF PPOL [PORT {numeric expression}]

ON PPOL (On Parallel POLl) causes periodic parallel polls to be performed on the specified port. All devices on the port are polled simultaneously. If no port is specified, port 0 is assumed. Any parallel polling in process is halted by any ON GOTO interrupt. The RESUME statement causes parallel polling to continue.

- ON PPOL enables parallel polling.
- OFF PPOL disables parallel polling.

If the result of a poll is not zero, control is passed to the specified line number. The line number indicates the beginning of a parallel poll handling routine. RESUME returns control to the next statement after the one completed when the interrupt occurred.

PORTSTATUS() Function

Usage: PORTSTATUS(port p%)

The PORTSTATUS() function is provided to permit a program to determine the status of an interface port. The PORTSTATUS function returns the port status code (of type integer) for port p%. An error will be reported if the value of p% is outside the range of [0..1]. In the case of a floating point p%, the value will be truncated to an integer value. An overflow error may be reported when this truncation is performed.

PPL() Function

Usage: PPL(port p%)

PPL (Parallel Poll) performs a parallel poll of a specified instrument bus port and returns an integer result between 0 and 255.

The correspondence between the IEEE-488 DIO lines and binary bit numbers is as follows:

DIO Line	Bit Number	Numeric Weight
DI01	0	1
DI02	1	2
DI03	2	4
DI04	3	8
DI05	4	16
DI06	5	32
DI07	6	64
DI08	7	128

SPL Function

Usage: SPL(device number)

SPL (Serial PoLl) performs a serial poll of a specified instrument and returns an integer status byte result between 0 and 255. By sequentially performing serial polls of instruments and checking for SRQ in the status bytes, the SRQ routine can determine which instruments set SRQ. By examining the remaining bits of some instruments, the SRQ routine can determine why and take appropriate action.

WAIT Statement

Usage:
```
WAIT [time expression] [FOR KEY]
WAIT [time expression] [FOR PPOL]
WAIT [time expression] [FOR SRQ]
WAIT [time expression] [FOR TIME]
WAIT FOR TIME
```

WAIT suspends program execution until either the specified time period elapses or until an enabled interrupt occurs. By specifying an event to wait for (KEY, PPOL, or SRQ), an interrupt can be enabled for only the duration of the waiting period. (See the Touch- Sensitive Display section for a discussion of KEY.)

Wait time is indefinite if a time is not specified. The minimum wait time is 10 milliseconds and the clock resolution is 10 milliseconds. The time given in a time expression is specified according to one of the following formats:

hh:mm:ss	Hours, minutes, seconds (24 hours maximum).
hh:mm	Hours and minutes.
milliseconds	Up to 86400000 milliseconds (24 hours).

If the event specified is FOR TIME, the Controller will wait unconditionally for the time interval specified by the numeric expression to elapse.

When an interrupt that has been enabled by an ON statement occurs, the program branches to the interrupt handling routine. If the interrupt occured prior to the end of the WAIT period, a subsequent WAIT FOR TIME statement will wait for the remainder of the WAIT period. A RESUME will execute the statement following the WAIT.

Section 10
RS-232 Serial I/O

CONTENTS

Introduction 10-3
Overview 10-3
RS-232-C Defined 10-4
Device and Port Addressing 10-5
  Initialization 10-6
  I/O Channels 10-6
    OPENing a Serial Communication Channel 10-7
    CLOSEing a Serial Communications Channel 10-8
Output and Input 10-9
  PRINT Statement 10-9
  INPUT Statement 10-10
  INCOUNT() Function 10-12
  INCHAR() Function 10-13
  Sending BREAK 10-14
Establishing Serial Communications 10-15

INTRODUCTION

The serial I/O ports on the Instrument Controller use the Electronic Industry Association's RS-232-C Data Communications Interface Standard (RS-232).

RS-232 is the standard used by many devices that use serial data communications to pass information. The RS-232 standard describes the physical connector, the signals on each pin of the connector, timing requirements, and the voltage levels of the signals.

RS-232 type devices include (and certainly aren't limited to) printers, modems, and data terminals.

OVERVIEW

This section describes the BASIC commands used for RS-232 input and output via the serial I/O ports on the Instrument Controller.

RS-232-C DEFINED

EIA Standard RS-232-C (RS-232-C will be referred to as "RS-232" for the balance of this section) provides the electronics industry with the ground rules necessary for independent manufacturers to design and produce both data terminal and data communication equipment that conforms to a common interface requirement. As a result, a data communications system can be formed by connecting an RS-232 data terminal to an RS-232 data communication peripheral (such as a teletype, modem, computer, etc.).

RS-232-C is a hardware standard which guarantees the following:

1. Each device using the RS-232 standard will use a standard 25-pin connector which will mate to another standard 25-pin connector of the opposite sex.

2. No matter how the cables are connected, no smoke or damage will occur.

3. The data and handshaking lines will each be given a specific name.

Additional information on RS-232 communications may be found in the following publications:

1. 1722A System Guide, Section 5, "Serial Communications".

2. Fluke Application Information #B0101, "1720A RS-232-C Interfacing to Serial Printers".

3. EIA Standard RS-232-C, "Interface Between Data Terminal Equipment and Data Communications Equipment employing Serial Binary Data Interchange", available from:

   Electronic Industries Association
   Engineering Department
   2001 1st Street, Northwest
   Washington D.C. 20006

4. "Industrial Electronics Bulletin No 9 - Application Notes for EIA Standard RS-232-C", also available from the EIA.

DEVICE AND PORT ADDRESSING

The Instrument Controller has several devices that use serial communications for data transfer between themselves and the Controller. These devices and their device designations are:

DEVICE NAME	DESIGNATION
Display and Programmer Keyboard	KB0:
RS-232 Serial Port 1	KB1:
RS-232 Serial Port 2 (optional)	KB2:
17XXA-009 Dual Serial Interface	SP0: - SP9:

These devices differ from the other devices used in and with the controller: they are not file-structured devices. To avoid confusion, remember which devices are file-structured and which ones are not. Section 4 of the System Guide discusses devices and files.

Initialization

Before any input or output can take place from a serial port, it must be initialized. This means that the data transmission rate, data format, transmission protocols, parity, and other parameters must be set to match those of the peripheral device. The SET utility program performs this function. Run the SET program from BASIC by typing:

```
EXEC "set"
```

or from FDOS:

```
FDOS> set
```

The SET program is fully described in the System Guide.

I/O Channels

The Instrument Controller communicates between the BASIC program and various devices by means of I/O channels.

There can be a maximum of 16 I/O channels open at any one time. They are designated with the numbers 1 through 16. A device must be associated with a channel for any data transfer to occur.

The I/O channels are also used to communicate with instruments connected to the IEEE-488 bus as well as file-structured devices. Refer to sections 7 and 9 of this manual.

OPENing a Serial Communication Channel

Usage: OPEN {device designation} AS OLD FILE {channel number}

This form of the OPEN statement is used to associate a device with a channel number for serial input. The OPEN statement is also used to associate a file with a channel number.

Usage: OPEN {device designation} AS NEW FILE {channel number}

Use this form of the OPEN statement to establish a channel for serial output.

The following points apply to both usages of the OPEN statement for serial I/O.

- All subsequent input from and output to the device is made by reference to the channel number.
- The channel is sequential access. Data is sent to, or retrieved from, sequential (serial) channels in serial order.
- The device designation string following OPEN indicates the name of the device.

The AS NEW and AS OLD clauses of the OPEN statement indicate specific actions for sequential channels. The following points discuss these differences:

- AS must be specified. If it is not followed by NEW or OLD, OLD is assumed.
- NEW indicates an output channel (from the Controller, to the device). OLD, or no specification, indicates an input channel (from the device to the Controller).

The numeric expression following FILE indicates the channel number to be assigned. Following are some points to be considered in selecting a channel number:

- The value of the numeric expression must be between 1 and 16.
- Each channel number can only be used for one operation (input or output) at a time.
- A channel that was previously opened for a different purpose, and is no longer in use, must first be closed before being reassigned.

CLOSEing a Serial Communications Channel

Usage: CLOSE ALL
CLOSE [numeric expression]

The CLOSE statement frees a previously opened channel for other use. As part of this process, some specific actions are taken:

- The input or output of data in memory to or from the specified channel is first completed.
- Interrupts are disabled for the channel, if it was opened for input from a serial (RS-232-C) port and interrupts had been enabled. This is equivalent to an OFF #n statement in addition to closing the channel.
- An End-of-File mark (CTRL/Z character) is then sent, if the channel was opened for sequential output.
- All opened channels can be closed by the CLOSE ALL statement. Channel numbers are separated by commas.

OUTPUT AND INPUT

The PRINT and INPUT statements are used to communicate with system level serial devices through a previously opened channel. The PRINT statement allows raw or formatted output to a printer or other device. The INCOUNT() and INCHAR() functions are also described in this section.

PRINT Statement

Usage: PRINT {#n,}[USING {format description,} string$; or , string2$]

The PRINT statement is described here for output to a device through a previously opened channel. PRINT may also be used for direct display output, or for output to instruments on the IEEE-488 bus. These applications of PRINT are described in other sections in this manual.

- The numeric expression following PRINT selects a previously opened channel. See the OPEN statement described in this section.
- The items to be printed are listed, separated by either a comma or a semicolon.
- The items to be printed may include integer, floating point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to Section 6 of this manual for a discussion of array subranges.
- When the open channel is to a serial (RS-232-C) port, the End-Of-File character transmitted will be as previously defined by the SET Utility program.
- The USING option may be specified for formatted output. Refer to the Reference Section for further details.

INPUT Statement

Usage: INPUT[#n,] [LINE] {variable list}

INPUT is described here for data input from a device through a previously opened channel. INPUT may also be used for direct keyboard input, or for input from instruments on the IEEE-488 bus. These applications of INPUT are described in other sections in this manual.

- The optional LINE specification is discussed in the Reference Section of this manual.
- The numeric expression following INPUT or INPUT LINE selects a previously opened channel. See the OPEN statement described in this section.
- The variables that will store the data input are listed, separated by a comma.
- The input data may include integer, floating point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to Section 6 of this manual for a discussion of array subranges.
- When the open channel is from a serial (RS-232) port, incoming Carriage Return and Line Feed characters are deleted. A Carriage Return, Line Feed sequence is appended after each occurrence of the line terminator character defined by the SET RS-232 Utility program. If Line Feed or Carriage Return is the line terminator, this process does not duplicate it.
- When the open channel is from a serial (RS-232) port, the End-of-File character defined by the SET RS-232 Utility program is deleted, and ⟨CTRL/Z⟩, CHR$(26), is put in its place.
- BASIC does not send a prompt character, "?", when input is from a channel.
- If the timeout expires for a device as initialized using the SET utility, an error 300 will be generated.
- The input statement is described in greater detail in the BASIC Reference manual.

In the following example a sequential input channel from the keyboard is opened. When lines 110 and 120 are executed, the message displayed is the string at line 110. This technique allows an INPUT statement, such as line 120, to be used without the usual "?" prompt.

```
10   OPEN "KB0:" AS OLD FILE 2
20   ! Other statements
30   !
110  PRINT "ENTER THE SERIAL NO:   ";
120  INPUT #2, SN$
130  ! Other statements
```

In the following example, assume a device is attached to RS-232 Port 1 which can print data sent to it and send data to the Instrument Controller. Lines 20 and 30 assign KB1: simultaneously for input and output, using separate channels. This program can send prompts on the output channel (line 100) and receive data on the input channel (line 120). An example of such a device is a printing computer terminal. Note that a "?" is not sent at line 100. This simultaneous assignment for input and output is not possible for a sequential channel to a file.

```
10   REM -- Demonstrate Input and Output From Same Device
20   OPEN 'KB1:' AS OLD FILE 1      ! Input channel 1
30   OPEN 'KB1:' AS NEW FILE 2      ! Output channel 2
40   ! Other statements
50   !
100  PRINT #2, A$                   ! Give prompt
110  INPUT #1, A(0..5)              ! Get 6 values
120  ! Other statements
```

INCOUNT() Function

Format: INCOUNT(channel%)

The number of input characters and/or lines available from a serial device (console or RS-232 port) may be determined by the INCOUNT() (input count) function. In the example shown above, channel% is the channel number (0 for the console) for which the number of characters available is desired. The integer values returned by INCOUNT() are:

0	no characters available
<> 0	number of characters and/or lines available

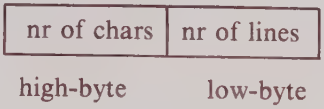
These values are returned as integers.

An I/O error 308 will be reported if either the channel specified is not open (note that the console, which is "channel" zero, is always considered "open"). An I/O error 322 will be reported if the device attached to the channel is not a serial (RS-232 or console) device.

The result returned by INCOUNT(0%), which is the amount of data available from the console, will be reported in terms of characters available if the console is in NOECHO mode (see the SET NOECHO statement). The result returned by INCOUNT(0%) will be reported in terms of lines available if the console is in ECHO mode (see the SET ECHO statement). An INPUT statement or INCHAR() function call directed to the console will be blocked until the value returned by INCOUNT() is not zero, which is to say:

1. NOECHO mode is active and a character has been typed, or
2. ECHO mode is active and a complete line has been typed.

The result returned for INCOUNT(n%), where n% (the channel number) is not zero, will be returned as a combined line and character count as follows:



The high- and low-bytes may be separated using the LSH function and a bit mask. A program example may be found in the Reference volume entry for INCOUNT.

An INPUT statement which is to read data from the channel will be blocked until the number of lines available is not zero. The INCHAR() function is blocked until the number of characters available is not zero.

INCHAR() Function

Format: INCHAR(channel%)
INCHAR(0%)

A single character may be read from any open channel or from the console by the INCHAR() function. In the example, channel% is the channel number from which the next character is desired. If channel% is zero the next character from the console will be returned. The character values are returned by INCHAR() are integers, and will have a value between 0 and 255 (0 and 127 when reading characters from the console).

The INCHAR() function is useful when input must be processed on a per-character basis or when data from the console keyboard is being entered using NOECHO mode (see the SET NOECHO statement). A BASIC program using INCHAR(0%) to read data from the console must be prepared to process line editing (⟨CTRL⟩/U and DELETE) characters as required by the application, since FDOS will not do so during single-character input.

Sending BREAK

Usage: BREAK {device%}

The BREAK statement sends a break signal to an RS-232 port. device% is the device number of an RS-232 port and is derived from the device name as follows:

device%	device name
1	KB1:
2	KB2:
3	KB3:
...	...
9	KB9:

Error 329 (illegal BREAK parameter) is reported if the value of device% is outside of the range of device% $\geq$ 1 and device% $\leq$ 9.

ESTABLISHING SERIAL COMMUNICATIONS

Follow these steps for communicating with one of the serial devices.

1. Use the OPEN statement to establish an input or output channel. It is good practice to CLOSE an I/O channel before opening it.

2. Use the PRINT (USING) #n statement to send data to a serial output channel.

3. Use the INPUT (LINE) #n statement to receive data from a serial input channel.

4. Use the CLOSE statement to "disconnect" the I/O channel from the BASIC program. The CLOSE statement is also used before re-assigning a device to another channel.

5. Before running your program, use the SET Utility Program to configure the serial port(s) to your device(s). This can be done at turn-on via the Startup Command file. Both the SET utility and the Startup Command file are described in the System Guide.

Section 11
Interrupt Processing

CONTENTS

Introduction 11-3
Overview 11-3
Types of Interrupts 11-4
  Error Interrupt 11-4
  ⟨CTRL⟩/C Interrupt 11-4
  #n Interrupt 11-5
  KEY Interrupt 11-5
  PORT Interrupt 11-5
  PPORT Interrupt 11-5
  SRQ Interrupt 11-5
  PPOL Interrupt 11-5
  CLOCK Interrupt 11-5
  INTERVAL Interrupt 11-5
Hierarchy of Interrupts 11-6
On-Event Interrupts 11-7
  ON ERROR GOTO Statement 11-8
  OFF ERROR Statement 11-9
  ON CTRL/C GOTO Statement 11-10
  OFF CTRL/C Statement 11-11
  ON #n GOTO Statement 11-12
  OFF #n Statement 11-13
  ON KEY GOTO Statement 11-14
  OFF KEY Statement 11-15
  ON PORT Statement 11-15
  OFF PORT Statement 11-15
  ON PPORT Statement 11-15
  OFF PPORT Statement 11-15

ON SRQ GOTO Statement 11-16
OFF SRQ Statement 11-17
ON PPOL GOTO Statement 11-18
OFF PPOL Statement 11-19
SET CLOCK Statement 11-19
ON CLOCK Statement 11-19
OFF CLOCK Statement 11-19
SET INTERVAL Statement 11-20
ON INTERVAL Statement 11-20
OFF INTERVAL Statement 11-20
RESUME Statement 11-21
WAIT FOR EVENT Interrupts 11-22
WAIT Statement 11-24
WAIT Time Statement 11-24
WAIT FOR KEY Statement 11-25
WAIT FOR SRQ Statement 11-26
WAIT FOR PPOL Statement 11-27
Errors and Error Handling 11-28
Fatal Errors 11-28
Recoverable Errors 11-28
Warning Errors 11-29
Error Variables 11-29
Interrupt Control Statements 11-30
Interrupt Processing Program Examples 11-31

INTRODUCTION

This section describes statements and functions which enable and process interrupts to a BASIC program. Interrupts are a response to events that may occur during normal program execution. For example, if the operator determines that an instrument test is not performing properly because of an outside influence (such as the IEEE-488 Bus cables not connected), the ABORT switch can be pressed to terminate the test. Since an operator may not have access to the programming keyboard, the test program must have the ability to analyze conditions, to make appropriate responses, and to restart itself. Event interrupt processing tasks are determined by the individual requirements of the system. Refer to Section 9 for further information on IEEE-488 Bus Polling statements.

OVERVIEW

Interrupt processing is discussed in four subject areas. First is a discussion of the types of interrupts, their priorities and their heirarchies. Second, the techniques of enabling the Instrument Controller to respond with a predefined section of program any random time that an interrupting event occurs are discssed (ON EVENT interrupts). Next is a discussion of the method of stopping program execution at a particular point to wait for an event to occur before proceeding (WAIT FOR EVENT interrupts). The final subject area is a discussion of the handling of errors (Error Handling). Program examples illustrate the concepts used.

TYPES OF INTERRUPTS

Fluke BASIC recognizes nine types of interrupts. Interrupts are recognized in order of their priority. These are listed with their priority in Table 11-1.

Table 11-1. On-Event Statements

PRIORITY	INTERRUPT TYPE
1	ON ERROR
1	ON CTRL/C
2	ON #n (RS-232-C Channel)
3	ON KEY (the Touch-Sensitive Display)
4	ON PORT
5	ON PPORT
6	ON SRQ (IEEE-488 Bus instrument service request)
7	ON PPOL (IEEE-488 Bus parallel poll)
8	ON CLOCK
9	ON INTERVAL Interrupts

Error Interrupt

An error interrupt occurs when errors are detected during program execution. Errors are divided into three levels that differentiate the types of response possible.

- Fatal — Immediately terminates the program.
- Recoverable — Statement processing stops immediately then terminates the program unless acknowledged by an error interrupt.
- Warning — Program continues running even if error interrupt is not enabled.

⟨CTRL⟩/C Interrupt

A ⟨CTRL⟩/C interrupt may be initiated from either of two sources:

- Pressing the ABORT button on the front panel.
- Entering ⟨CTRL⟩/C on the programmer keyboard.

#n Interrupt

A #n interrupt occurs whenever a line or file terminator character, as defined by the SET RS-232 Utility program, is received through an open input channel from a serial (RS-232-C) port. The number following "#" is the channel number.

KEY Interrupt

A KEY interrupt occurs whenever the Touch-Sensitive Display is touched. The ON KEY statement causes an immediate branch to a specified line number.

PORT Interrupt

A PORT interrupt occurs when a change in the state of an IEEE-488 bus port occurs (for example, being addressed as a Listener).

PPORT Interrupt

A PPORT interrupt occurs when a device connected to a parallel I/O port toggles its handshake line to request an interrupt.

SRQ Interrupt

An SRQ interrupt occurs when an instrument on the selected IEEE-488 port issues a service request.

PPOL Interrupt

A PPOL interrupt occurs when the parallel poll response from instruments on the selected IEEE-488 Bus port is a non-zero value.

CLOCK Interrupt

A CLOCK interrupt occurs as a result of the ON CLOCK statement. The interrupt occurs at the time set by the SET CLOCK statement.

INTERVAL Interrupt

A INTERVAL interrupt occurs as a result of the ON INTERVAL statement. The interrupt occurs at the time set by the SET INTEVAL statement.

Hierarchy of Interrupts

Interrupts have a hierarchial relationship that avoids conflicts when two or more interrupts become simultaneously active.

- ERROR and ⟨CTRL⟩/C can interrupt each other and share top priority. They preempt acknowledgement of other interrupts.
- Other interrupts that subsequently occur are not recognized until after the ERROR or ⟨CTRL⟩/C is acknowledged.
- There is one difference between ERROR and ⟨CTRL⟩/C.

  1. If a second ERROR occurs before the first is acknowledged, the program terminates immediately.
  2. If a second ⟨CTRL⟩/C occurs before the first is acknowledged, the second is ignored.

- Second priority is #n.
- Third priority is KEY.
- Fourth priority is PORT.
- Fifth priority is PPORT.
- Sixth priority is SRQ.
- Seventh priority is PPOL.
- Eigth priority is CLOCK.
- Ninth priority is INTERVAL.
- The #n, KEY, PORT, PPORT, SRQ, PPOL, CLOCK and INTERVAL interrupts do not preempt each other. However, they may be preempted by either ERROR or ⟨CTRL⟩/C.

ON-EVENT INTERRUPTS

On-event interrupts enable the Instrument Controller to respond to events that occur at random times that cannot be known when the program is written.

- There are three actions that occur with on-event interrupts:

  1. If an ERROR or ⟨CTRL⟩/C interrupt occurs during the processing of an interrupt, the program is stopped or control is transferred as required by the ERROR or ⟨CTRL⟩/C.

  2. If any other interrupts occur during the processing of an interrupt, they are assigned a pending status until a RESUME statement is encountered.

  3. When a RESUME statement is processed, control is transferred to the highest priority pending interrupt or to the next statement following the one that was in process when the interrupting event occurred.

- Interrupts are initially disabled.

- The interrupt must first be enabled (by ON event GOTO line number) to allow the interrupting event to redirect the program sequence.

- After completing a response to the interrupt that occurred, the interrupt must be acknowledged with a RESUME statement.

- If interrupt response is no longer required, it may be disabled with an OFF statement.

- Any or all interrupting conditions may be activated simultaneously.

- Table 11-2 summarizes these actions.

Table 11-2. On-Event Statements

ACTION	STATEMENT
Enable	ON interrupt name GOTO line number
Acknowledge	RESUME (line number)
Disable	OFF interrupt name

ON ERROR GOTO Statement

Usage: ON ERROR GOTO {line number}

The ON ERROR GOTO statement enables a program to respond to a random occurrence of an error condition by transferring control to a specified routine containing a user-defined response.

- When an error is detected, control transfers to the specified line number immediately.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- Only level R and W errors can be processed by an error routine. Level F errors always terminate the program.
- Level R interrupts will terminate a program unless an ON ERROR GOTO statement has been executed so that the error condition can be treated.
- Without error processing, a level W error is ignored.
- When an error condition has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until a RESUME is executed.
- When an error condition has been detected, the system variable ERL will contain the line number at which the error occurred, and ERR will contain the error number.
- If a second error is detected before encountering a RESUME statement, the program terminates immediately.

NOTE

Further interrupt processing except ⟨CTRL⟩/C is suspended until a RESUME statement is executed.

NOTE

A RESUME statement without a line number will re-execute the statement that caused the error.

- If a ⟨CTRL⟩/C interrupt occurs after error detection and before encountering a RESUME statement, error processing is suspended either temporarily or permanently. If the program includes ⟨CTRL⟩/C interrupt processing with a RESUME statement, control will be returned to the error processing routine when ⟨CTRL⟩/C processing is completed. See the ON ⟨CTRL⟩/C GOTO statement in this section.

OFF ERROR Statement

Usage: OFF ERROR

The OFF ERROR statement disables the action of a previous ON ERROR GOTO statement.

- An OFF ERROR statement in an error processing routine will terminate the program.
- After an OFF ERROR statement has been executed, a level R error will terminate the program.
- After an OFF ERROR statement, a level W error will be ignored.

ON CTRL/C GOTO Statement

Usage: ON CTRL/C GOTO {line number}

The ON CTRL/C GOTO statement enables a program to respond to a random occurrence of an ABORT switch or a ⟨CTRL⟩/C keyboard entry by transferring control to a specified program routine containing a user defined response.

- When an ABORT switch or a ⟨CTRL⟩/C keyboard entry is detected, control transfers to the specified line number.
- When the SET NOECHO statement has put the keyboard into "character mode" a ⟨CTRL⟩/P keyboard entry will transfer control to the line number specified by ON CTRL/C.
- The ⟨CTRL⟩/C handling routine must explicitly acknowledge the interrupt with a RESUME statement.
- BASIC normally responds to the ABORT switch or a ⟨CTRL⟩/C keyboard input by terminating the program and returning to Immediate Mode. This statement alters the normal interpreter response.
- When a ⟨CTRL⟩/C has been detected, further checking for interrupt conditions other than ERROR is suspended until RESUME is encountered.
- If a second ⟨CTRL⟩/C or ABORT is detected before encountering a RESUME statement, it is ignored.
- A RESUME statement will return control to execute the first statement not completed when the ⟨CTRL⟩/C or ABORT key entry was detected.

NOTE

Further interrupt processing except ON ERROR is suspended until a RESUME statement is performed.

- If a level R or W error occurs after ⟨CTRL⟩/C detection and before encountering a RESUME statement, ⟨CTRL⟩/C processing is suspended either temporarily or permanently. If the program includes error processing, control will be returned to the ⟨CTRL⟩/C processing routine when error processing is completed. Without error processing, a level W error is ignored. See the ON ERROR GOTO statement in this section.

NOTE

Since ⟨CTRL⟩/C is the only way to manually stop a BASIC program without deleting it, if not handled properly, a ⟨CTRL⟩/C interrupt to an ON CTRL/C subroutine can lock a program into Run Mode.

OFF CTRL/C Statement

Usage: OFF CTRL/C

The OFF CTRL/C statement disables the action of a previous ON CTRL/C GOTO statement.

An OFF CTRL/C statement in a ⟨CTRL⟩/C interrupt processing routine will cause the Instrument Controller to return to Immediate Mode.

After an OFF CTRL/C statement, the ABORT switch or a ⟨CTRL⟩/C keyboard entry will return the controller to Immediate Mode.

ON #n GOTO Statement

Usage: ON #{channel%} GOTO {line number}

The ON #n GOTO statement enables a program to respond to End-Of-File characters received through a open input channel from a serial (RS-232-C) port, control is transferred to the specified program routine containing a user defined response whenever either terminator character is received.

- When a terminator character is received through a open input channel from a serial (RS-232-C) port, control is transferred to the specified line number after completion of the current statement.
- End-Of-Line and End-Of-File terminator characters are defined by the SET utility program. Refer to the Instrument Controller User Manual, November 1980 Revision (or later), for information.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement. Program control then resumes at the next statement following the one that was completed when the terminator character was received.
- When a terminator character has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until RESUME is encountered.
- If this statement is used more than once in a program with the same channel number, control is transferred to the line number referenced in the most recently encountered ON #n GOTO statement.
- Any of the 16 available user channels may be used. Refer to the OPEN statement discussion is the Reference volume of this manual set.
- The referenced channel must be opened for input (OLD or not specified) prior to the ON #n GOTO statement.
- When interrupts occur on more than one channel simultaneously, the lowest numbered channel has highest priority.

OFF #n Statement

Usage: OFF #{channel%}

The OFF #n statement disables the action of all previous ON #n GOTO statements for the referenced channel.

- An OFF #n statement in a serial port interrupt processing routine will disable further interrupts on the referenced channel, but will not affect the interrupt processing that is in progress.
- The OFF #n statement does not close the referenced channel.
- If the channel is closed by a CLOSE n statement, further interrupts are disabled and the OFF #n statement is unnecessary.

ON KEY GOTO Statement

Usage: ON KEY GOTO {line number}

The ON KEY GOTO statement enables a program to respond to the occurrence of a key entry on the touch-sensitive display by transferring control to a specified program routine containing a user defined response.

- When a key entry is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- When a KEY entry has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until RESUME is encountered.
- When a KEY entry has been detected, the system variable KEY will contain the number of the last touch key pressed.
- The system variable KEY is set whenever the Touch-Sensitive Display is pressed in an active area, regardless of whether ON KEY GOTO is used.

NOTE

Once the Touch-Sensitive Display is pressed in an active area, the KEY variable remains set until it is read by a program statement. (For example, K% = KEY) Once the KEY variable is read by a program statement, its value is reset to zero.

- If the system variable KEY is non-zero when ON KEY GOTO is executed, control is immediately transferred to the specified line number.
- ON KEY does not reset the system KEY variable.

OFF KEY Statement

Usage: OFF KEY

The OFF KEY statement disables the action of a previous ON KEY GOTO statement.

- An OFF KEY statement in a key interrupt processing routine will prevent the routine from being continuously reentered if the KEY buffer is not reset in the routine.
- An OFF KEY statement in any interrupt processing routine will not have any additional effect.

ON PORT Statement

Usage: ON PORT p% GOTO line number

The ON PORT statement permits the Instrument Controller to detect commands sent to it by another controller. Port 0 is assumed if p% is not specified. If p% is a floating-point variable, the value is rounded to its integer value. An overflow error may result from this action. Use the PORTSTATUS statement to determine the exact cause of the interrput.

OFF PORT Statement

Usage: OFF PORT p%

The OFF PORT statement disables a previous ON PORT interrupt. p% must be in the range of 0..1. It is an error for p% to be outside of this range. If p% is omitted, port 0 is assumed.

ON PPORT Statement

Usage: ON PPORT port% GOTO line number

The ON PPORT statement enables a program to respond to data from the optional Parallel Port by transferring control to a specified program routine containing a user-defined response. If port% is not specified, port 0 is used.

OFF PPORT Statement

Usage: OFF PPORT port%

The OFF PPORT statement disables a previous ON PPORT statement. If port% is not specified, port 0 is used.

ON SRQ GOTO Statement

Usage: ON SRQ GOTO {line number}

The ON SRQ GOTO statement enables a program to respond to the occurrence of a service request from an instrument by transferring control to a specified program routine containing a user defined response.

- The specified port is sampled after the completion of each statement. If a port is not specified, port 0 is sampled.
- When a service request is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- When a service request has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until RESUME is encountered.
- An internal SRQ flag is set by a service request on the enabled port. It is reset in the controller by performing a serial poll on any instrument on the port requesting service (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably be set again until the instrument requesting service is serial polled. This will cause the service request routine to be immediately reentered after the RESUME statement.
- ON SRQ GOTO does not reset the internal SRQ flag.
- When SRQs are present on both port 0 and port 1 simultaneously, the SRQ on post 0 will be responded to first.

OFF SRQ Statement

Usage: OFF SRQ

The OFF SRQ statement disables the action of a previous ON SRQ GOTO statement.

- An OFF SRQ statement in a service request processing routine will prevent the routine from being continuously reentered if it does not reset the service request by performing a serial poll.
- An OFF SRQ statement in any interrupt processing routine will not have any additional effect.

ON PPOL GOTO Statement

Usage: ON PPOL GOTO {line number}

The ON PPOL GOTO statement enables a program to respond to a positive parallel poll response from a configured instrument by transferring control to a specified program routine containing a user defined response.

- The ON PPOL GOTO statement initiates parallel polling on the specified port, or on Port 0 if not specified. A poll will be performed following the completion of each BASIC statement.
- When a non-zero response to a parallel poll is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- When a non-zero response to a parallel poll has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until RESUME is encountered.
- If both Port 0 and Port 1 have PPOL interrupts enabled, port 0 will be checked for a parallel poll response prior to checking Port 1.

NOTE

Some instruments clear a parallel poll bit when the condition causing it disappears, or when the bus port is parallel polled.

When possible, the instrument responding to the parallel poll should be programmed within the processing routine to reset its poll response bit. If this bit remains set, the routine will be immediately reentered after the RESUME statement.

OFF PPOL Statement

Usage: OFF PPOL

The OFF PPOL statement disables the action of a previous ON PPOL GOTO statement.

- An OFF PPOL statement in a parallel poll response routine will prevent the routine from being immediately reentered after the RESUME statement if the routine does not clear the instrument poll response bit.
- An OFF PPOL statement in any interrupt processing routine will not have any additional effect.

SET CLOCK Statement

Usage: SET CLOCK {time expression}

The SET CLOCK statement is used to indicate (set) the time to be used for a timer interrupt. The SET CLOCK statement must be executed before an ON CLOCK statement, or error 708 (timer not set) will occur.

- The time expression used may denote the interval time in any of the following formats:

hh:mm:ss	Hours, minutes, seconds (24 hours maximum).
hh:mm	Hours and minutes.
milliseconds	Up to 86400000 milliseconds (24 hours).

ON CLOCK Statement

Usage: ON CLOCK GOTO {line number}

The ON CLOCK statement is used to enable an interrupt that occurs at a specific time of day. The SET CLOCK statement determines the time-of-day that the interrupt occurs. The time reference for the ON CLOCK statement is the System Clock built into the Controller.

OFF CLOCK Statement

Usage: OFF CLOCK

The OFF CLOCK statement disables a previously set ON CLOCK interrupt.

SET INTERVAL Statement

Usage: SET INTERVAL {time expression}

The SET INTERVAL statement is used to indicate (set) the interval between timer interrupts. This statement must be used before a corresponding ON INTERVAL statement or error 706 (timer not set) will occur.

- The time expression used may denote the interval time in any of the following formats:

hh:mm:ss	Hours, minutes, seconds (24 hours maximum).
hh:mm	Hours and minutes.
milliseconds	Up to 86400000 milliseconds (24 hours).

ON INTERVAL Statement

Usage: ON INTERVAL GOTO {line number}

The ON INTERVAL statement is used to enable an interrupt that occurs at an interval chosen by the programmer. The SET INTERVAL statement determines the interval used by the ON INTERVAL statement. The time reference for the ON INTERVAL statement is the System Clock built into the Controller.

OFF INTERVAL Statement

Usage: OFF INTERVAL

The OFF INTERVAL statement is used to disable a previously set ON INTERVAL interrupt.

RESUME Statement

Usage: RESUME [line number]

The RESUME statement acknowledges an interrupt and allows program operation to resume with the next statement after the one being completed when the interrupt occurred or at another specified program location.

- RESUME (no line number) branches to the statement following the one being executed at the point where the interrupt occurred.
- If the interrupt occurred in a multiple statement line, the program resumes with the next statement on the line.
- There are two exceptions:
  1. Recoverable errors: The program resumes at the beginning of the statement that caused the error.
  2. Input Warning errors 801, 802, and 803: The INPUT statement which caused the error requests the value to be entered again. It did not accept the erroneous entry.
- RESUME (line number) branches to the specified line number.
- RESUME terminates the interrupt handler routine.

WAIT FOR EVENT INTERRUPTS

The WAIT (time) (FOR event) statement suspends program execution until the specified interrupt event occurs or the specified time elapses.

- The interrupt is implicitly acknowledged by its occurrence.
- WAIT may be followed by a numeric expression specifying a maximum period of time to wait for the interrupt.
- When the specified time elapses, interrupt checking stops and the program continues with the next statement.
- An interrupt previously enabled by an ON-GOTO statement remains enabled during the waiting period, whether or not the WAIT statement references it.
- The time interval may be specified in any of the following ways:

hh:mm:ss	Hours, minutes, seconds (24 hours maximum).
hh:mm	Hours and minutes.
milliseconds	Up to 86400000 milliseconds (24 hours).

- WAIT interrupts have four forms as shown in Table 11-3 with their meanings. Each construction of the WAIT statement is separately discussed below.

Table 11-3. WAIT Interrupt Statements

STATEMENT FORM	MEANING
WAIT	Suspend program execution until a ⟨CTRL⟩/C or an interrupt enabled by ON-GOTO occurs.
WAIT numeric expression	Suspend program execution up to the specified time limit until ⟨CTRL⟩/C or an interrupt enabled by ON-GOTO occurs.
WAIT FOR (KEY)(,)(PPOL)(,)(SRQ)	Suspend program execution until ⟨CTRL⟩/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.
WAIT numeric expression FOR (KEY)(,)(PPOL)(,)(SRQ)	Suspend program execution, up to the specified time limit, until ⟨CTRL⟩/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.
WAIT FOR TIME	Wait for remainder of time from the last WAIT statement.
WAIT {numeric expression} FOR TIME	Non-interuptible WAIT statement.

WAIT Statement

Usage: WAIT

The WAIT statement suspends operation of the program indefinitely.

- The ABORT switch, or a ⟨CTRL⟩/C keyboard entry, will always terminate the WAIT.
- If an ON CTRL/C GOTO statement has been executed, the ABORT switch or a ⟨CTRL⟩/C keyboard entry will terminate the WAIT and transfer control to the ⟨CTRL⟩/C processing routine.
- The WAIT statement takes the place of the (not allowed) construct WAIT FOR CTRL/C, since this top priority interrupt always remains enabled.
- A KEY, serial port, SRQ, or PPOL interrupt will not terminate the WAIT unless previously enabled by ON-event GOTO.

WAIT Time Statement

Usage: WAIT [time expression]

The WAIT Time statement suspends operation of the program for the specified length of time.

- Time may be specified in any of the following methods:

hh:mm:ss	Hours, minutes, seconds (24 hours maximum).
hh:mm	Hours and minutes.
milliseconds	Up to 86400000 milliseconds (24 hours).

- Timer resolution is 10 milliseconds.
- A negative time value results in a waiting time of 0.
- Until the specified time period has elapsed, the WAIT can be terminated in any of the ways defined above under the WAIT statement.

WAIT FOR KEY Statement

Usage: WAIT [time expression] FOR KEY

The WAIT FOR KEY statement suspends operation of the program indefinitely until the Touch-Sensitive Display is pressed in the active area.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.
- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.
- The WAIT is terminated whenever the Touch-Sensitive Display is pressed in the active area.
- A touch key input causes the program to continue with the next statement unless a previous ON KEY GOTO statement has been executed.
- When a previous ON KEY GOTO statement has been executed, a touch key input causes the program to transfer to the KEY processing routine.
- The system variable KEY is set whenever the Touch-Sensitive Display is pressed in an active area, regardless of whether WAIT FOR KEY is used. It remains set until it is read by a program statement. (For example, K% = KEY)
- Wait time is 0 if the system variable KEY is nonzero when WAIT FOR KEY is executed.

WAIT FOR SRQ Statement

Usage: WAIT [time expression] FOR SRQ

The WAIT FOR SRQ statement suspends operation of the program indefinitely until a service request is detected on either instrument port.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.
- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.
- The WAIT is terminated whenever a service request is detected on either instrument port.
- A service request causes the program to continue with the next statement unless a previous ON SRQ GOTO statement has been executed.
- When a previous ON SRQ GOTO statement has been executed, a service request causes the program to transfer to the service request processing routine.
- An internal SRQ flag is set by a service request. It is reset in the controller by performing any serial poll (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably not be set again until a serial poll is performed on the instrument requesting service.
- WAIT time is 0 if the internal SRQ flag was not reset after the last service request.
- WAIT FOR SRQ does not reset the internal SRQ flag.

WAIT FOR PPOL Statement

Usage: WAIT [time expression] FOR PPOL

The WAIT FOR PPOL statement suspends operation of the program and initiates continuous parallel polling indefinitely until a non-zero parallel poll response is detected on either instrument port.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.
- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.
- The WAIT is terminated whenever a non-zero parallel poll response is detected on either instrument port.
- A positive parallel poll response causes the program to continue with the next statement unless a previous ON PPOL GOTO statement has been executed.
- When a previous ON PPOL GOTO statement has been executed, a non-zero parallel poll response causes the program to transfer to the parallel poll processing routine.
- The program statements following WAIT FOR PPOL should cause the parallel poll response bit of the responding instrument to be reset, if possible.
- WAIT time is 0 if either instrument port has a non-zero parallel poll response when WAIT FOR PPOL is executed.
- See the note in the description of ON PPOL GOTO in this section.

ERRORS AND ERROR HANDLING

Section 17 and the Quick Reference Card list the errors which can occur in executing BASIC statements. There are three levels: Fatal, Recoverable, and Warning.

Fatal Errors

A fatal error (level F) terminates a program immediately, returns the Instrument Controller to Immediate Mode, and displays an error message.

- There is no way to recover a program from a level F error.

Recoverable Errors

A recoverable error (level R) terminates a program immediately, returns the Instrument Controller to Immediate Mode, and displays an error message.

- A level R error can be recovered from if:

  1. It occurs after execution of an ON ERROR GOTO statement.
  2. The program routine referenced by ON ERROR GOTO includes a RESUME statement.
  3. A second error does not occur before the RESUME.
  4. The program routine referenced by ON ERROR GOTO does not include an OFF ERROR statement.

- An OFF ERROR statement causes subsequent level R errors to terminate the program as described above.

Warning Errors

A warning error (level W) allows a program to continue running.

- An error message is normally displayed.
- An error message is not displayed if the four conditions described above under level R errors are met.
- Input warning errors 801 (too much data), 802 (not enough data), and 803 (illegal character), caused by keyboard entries, request that the entry be repeated. ("?" is displayed if input is from the display.)
- Error 803, caused by an illegal character in a VAL argument or a non-keyboard input, truncates the bad character and all characters following it. Characters up to the one causing the error are left intact.
- Error 904 (string longer than virtual array field) truncates all excess characters, leaving the rest intact.

Error Variables

There are two system variables for errors that may be used as variables in a program to determine what action to take when level R or W errors occur:

- ERL - the line number of the statement which caused the most recent error.
- ERR - the error number of the most recent error.

NOTE

ERL and ERR are reset when the RUN statement is executed but not when GOTO or CONT is executed in Immediate Mode.

INTERRUPT CONTROL STATEMENTS

The Interrupt Control statements allow the BASIC program to disable interrupts. The DISABLE statement disables all interrupts except for ON ERROR and CTRL/C. The ENABLE statement re-enables the interrupts after a DISABLE statement.

The DISABLE and ENABLE statements are used to isolate a "critical section" of code from an unwanted branch to an interrupt service routine.

While interrupts are never recognized during the execution of a single statement (except for CTRL/C), a critical section that extends across several statements must be able to disable interrupts to prevent erroneous updating of any shared variables.

Any interrupts which occur after the execution of a DISABLE statement are held pending the execution of an ENABLE statement.

INTERRUPT PROCESSING PROGRAM EXAMPLES

The program examples presented below illustrate the concepts developed in this section with portions of programs. These examples are not complete programs.

In the following example, the programmer anticipated that the variable I may be zero at some point and, rather than allow the program to halt, included an error handling routine that checks ERR for 603 (divide by zero error) and ERL for the line number of the possible divide by zero error. (Other divide by zero errors presumably should halt execution.) If the error or line number that caused the interrupt are not the ones expected, then the OFF ERROR disables the error handler routine and displays the error number and line number of the error. If error 603 did occur at line 120 to generate the interrupt, then the message is printed by line 1020. Line 1030 causes a branch to line 130, and the "ON ERROR GOTO 1010" is still active.

```
20      ON ERROR GOTO 1010
        .
        .
        .
110     . . .
120     A = 20 / I
130     . . . .
        .
        .
        .
1000    ! ERROR HANDLER
1010    IF ERR 603 OR ERL <> 120 THEN OFF ERROR
1020    PRINT "DIVIDE BY ZERO ERROR"
1030    RESUME 130
```

The following example illustrates an appropriate way to terminate a FOR-NEXT loop within a subroutine after resuming from an error handling routine. The FOR-NEXT loop was reset to its terminal value (-10) in the error handling routine so that the loop will not be repeated when control is resumed from the error handler. If control is resumed at line 4040 in the subroutine, error 521 (Illegal statement structure) results. In this example, the FOR-NEXT loop is allowed to terminate naturally.

```
30      ON ERROR GOTO 1010
        .
        .
        .
100     DIM Y (20)
110     GOSUB 4000
120     . . . .
        .
        .
        .
1000    ! ERROR HANDLER
1010    IF ERR <> 606 THEN OFF ERROR \  ! Negative LOG Argument
1020    PRINT "LOG ARGUMENT <= 0"
1030    I = -10
1040    RESUME 4030
        .
        .
        .
4000    ! CALCULATE AND STORE LOG ROUTINE
4010    FOR I = 10 TO -10 STEP -1
4020    Y (10 + I) = LOG (I)
4030    NEXT I
4040    RETURN
```

The following example uses a bus timeout (error 408, timeout during Bus I/O transfer) to terminate an input from the IEEE-488 Bus. When the sending device has ceased output to the bus, the 100-millisecond timeout set in line 100 generates an error that is handled in the error handling routine at line 1000 by first checking to see if it is the proper type of error. The error handling routine then determines which reading was the last reading (line 1020), resets the FOR-NEXT loop, and resumes executing at the NEXT statement. The statement GOTO 1050 at line 1010 causes a RESUME to re-execute any other statement that causes an error. When it is re-executed, the program terminates unless it is a warning error because the error will occur again.

```
10      ON ERROR GOTO 1010
        .
        .
        .
100     TIMEOUT 100                 ! Set BUS timeout to 100 msecs.
110     FOR I% = 0% TO 999%
120     INPUT LINE @ C3%, R$(I%)
130     NEXT I%
140     PRINT R$ (0%..N%)
150     STOP
        .
        .
        .
1000    ! TIMEOUT ERROR HANDLER
1010    IF ERR <> 408 THEN OFF ERROR
1020    N% = I% - 1%                ! N% = Number of readings
1030    I% = 999%                   ! Properly terminates FOR-NEXT
1040    RESUME 130
```

In the following example, RESUME returns control to the error causing statement rather than to a specified line number. The value of D is corrected in line 1010 so that the statement on line 30 will be re-executed properly.

```
10      ON ERROR GOTO 1000
20      D = 0
30      I = 3 / D
40      PRINT "I="; I
        .
        .
        .
1000    IF D < > 0 GOTO 32767       ! To END statement
1010    D = 1E-76
1020    RESUME
        .
        .
        .
32767   END
```

In the following example, D is not adjusted and the RESUME statement branches to a PRINT statement.

```
10      ON ERROR GOTO 1010
20      D = 0
30      I = 3 / D
40      PRINT "I="; I
50      PRINT "D="; D
        .
        .
        .
1000    ! ERROR ROUTINE
1010    IF D < > 0 GOTO 32767       ! To END statement
1020    RESUME 50
        .
        .
        .
32767   END
```

The following error routine checks for a valid input for a tangent value. The error is produced on line 30 when BASIC tries to find the tangent of the entered value. The likely error would be a divide by zero error. RESUME branches to line 20 to allow re-entry of the A value.

```
10      ON ERROR GOTO 100
20      PRINT "Enter value for TANGENT";/ INPUT A
30      X = TAN (A)
        .
        .
        .
100     IF ERR = 603 AND ERL = 30 THEN PRINT "ILLEGAL VALUE"
110     RESUME 20
```

The following example uses the ERR system variable to check for a number of possible error conditions and has the operator correct the problem by using the Touch-Sensitive Display.

```
10      INIT PORT 0                     ! Initialize Bus 0
20      ON ERROR GOTO 1010              ! Enable error interrupt
30      CLOSE 2                         ! Ensure file 2 not open
40      OPEN "DATA" AS NEW FILE 2       ! Open data file
45      PRINT "Enter the instrument's address"
50      INPUT AD%                       ! Enter device address
60      !
70      ! Command instrument to take readings
80      !
90      INPUT LINE @ AD%, R$            ! Input the reading
100     !
110     ! Other statements
120     PRINT #2, R$                    ! Save the reading on file 2
130     !
140     ! Other statements
150     !
900     STOP
910     !
920     !
1000    ! ERROR HANDLER
1005    !       Check for illegal entry
1010    IF ERR = 801 OR ERR = 803 GOTO 1040
1015    !       Check instrument address
1020    IF ERR <> 401 AND ERR < > 402 GOTO 1060
1025    !       Display operator message
1030    PRINT "Wrong Bus address: reenter"
1040    RESUME 50                       ! Re-execute line 50
1050    ! Check for good disk
1060    IF ERR <> 300 AND ERR < > 301 GOTO 1140
1070    IF ERR = 300 THEN PRINT "Load disk"; \ GOTO 1090
1080    PRINT "Remove write protect label";
1090    PRINT "Then touch display"      ! Display message
1100    K = KEY                         ! Clear key buffer
1110    WAIT FOR KEY                    ! Enable and wait for key
1111    ! interrupt
1120    K = KEY                         ! Clear key buffer
1130    RESUME                          ! Re-execute line 40
1140    OFF ERROR                       ! Terminate program on other
1145                                    ! errors
```

In the following example, if a ⟨CTRL⟩/C character is generated, the error routine beginning at line 5000 will take control. The operator is given the choice of continuing or halting execution of the program. The operator enters either 1 (continue) or 2 (halt). If neither 1 or 2 is selected, the screen redisplays the choices. If 1 is selected, the RESUME statement branches to the line that was being executed when the ⟨CTRL⟩/C character was encountered. If 2 is selected, the OFF CTRL/C statement halts execution of the program and returns to Immediate Mode.

```
10      ON CTRL/C GOTO 5010
        .
        .
        .
5000    ! ⟨CTRL⟩/C TERMINATION ROUTINE
5010    PRINT "SELECT ONE"/ PRINT
5020    PRINT SPACE$(21);"1 CONTINUE"
5030    PRINT SPACE4(21);"2 STOP"
5040    PRINT SPACE4(21);\INPUT C
5050    IF C <> 1 AND C <> 2 GOTO 5010
5060    IF C = 2 THEN OFF CTRL/C
5080    RESUME
```

In the following example, if ABORT or ⟨CTRL⟩/C is pressed, the ABORT routine at line 5010 is executed. The routine allows the operator to confirm the ABORT or ⟨CTRL⟩/C by selecting either 1 (continue) or 2 (halt) readings. In this way, an inadvertent ⟨CTRL⟩/C or ABORT need not necessarily halt readings.

```
10      ON CTRL/C GOTO 5010
        .
        .
        .
100     PRINT @ 3%, 'VR2?'  ! Take readings from instrument 3
110     INPUT @ 3%, V
120     PRINT V
130     GOTO 100                        ! Take another reading
        .
        .
        .
5000    ! ABORT BUS READING ROUTINE
5010    PRINT "SELECT ONE" \ PRINT "1 = CONTINUE"
5015    PRINT "2 = STOP"
5020    INPUT A
5030    IF A <> 1 AND A <> 2 GOTO 5010
5040    IF A = 1 THEN RESUME ELSE OFF CTRL/C
```

NOTE

Many of the preceeding examples used ⟨CTRL⟩/C to interrupt an IEEE-488 Bus operation. This interrupt method should be used with care since ⟨CTRL⟩/C aborts any bus I/O in progress at the time of the interrupt. Premature termination of bus I/O in this manner may leave instruments connected to the bus in an undefined state.

In this example, the WAIT statement requires that a 500 millisecond delay be performed before printing the words "FLUKE Instrument Controller". The statement on line 120 clears the screen, then the sequence is repeated until something, such as ⟨CTRL⟩/C, interrupts.

```
100     WAIT 500
110     PRINT CHR$ (27%); "[2J"     !Erase screen
120     PRINT "FLUKE Instrument Controller"
130     GOTO 100
```

The WAIT statement on line 110 allows a two-second delay between source generation (2V dc) from the 5100A (Fluke Calibrator) and measurement on the 8500A (Fluke Digital Multimeter). This allows the 5100A time to generate the source voltage before attempting to make a measurement with the 8500A.

```
90      D% = 2000%
100     PRINT @ 1%, '2V,N'          ! 5100A Programmed for 2V dc
110     WAIT D%
120     PRINT @ 2%, '?'
130     INPUT @ 2%, V               ! Reading from 8500A
140     PRINT V
```

The following example reads the system clock each time the display is touched. The Controller computes the difference and displays the elapsed time between the first and second time the display is touched. Note that the key buffer is initially cleared (line 30) to ensure a WAIT at line 70 and cleared again at line 90 and 140 for the same purpose.

```
10    ! **** TIMER ****
20    !
30    ! Clear key buffer
40    E$ = CHR$(27)+'['                 ! Escape sequence for display
50    PRINT E$; '2J'                    ! Clear display
60    PRINT CPOS (6, 26); "Touch to START"; ! Position cursor
61    K% = KEY                          ! and request start
70    WAIT FOR KEY                      ! Wait till display is touched
80    T1 = TIME                         ! Find time from system clock
90    K% = KEY                          ! Clear key buffer
95    ! Position cursor; display time 1
100   PRINT E$; 'J'; " T1 = "; T1;
110   PRINT CPOS(8, 26); "Touch to STOP "; ! Position cursor
111   ! and display stop request
120   WAIT FOR KEY                      ! Wait until display is touched
130   T2 = TIME                         ! Find time from system clock
140   K% = KEY                          ! Clear key buffer
150   PRINT "    T2 = "; T2             ! Display time 2
160   PRINT CPOS(10,32); "ELAPSED TIME = "; (T2-T1) / 1000;
165   PRINT "SECONDS"
170   GOTO 60                           ! Repeat program
```

Results from running this program:

```
Touch to START T1 =        2484420
Touch to STOP  T2 =        2489580

    ELAPSED TIME  =        5.16 SECONDS

Touch to START T1 =        2493550
Touch to STOP  T2 =        2493930

    ELAPSED TIME  =        0.38 SECONDS
```

Section 12
Subroutines

CONTENTS

Introduction 12-3
Overview 12-3
Using Internal Subroutines 12-4
Using External Subroutines 12-5
  Software Requirements 12-6
  Subroutine Names 12-6
  Assembly Language Subroutines 12-7
  Assembly Language Error Handler 12-7
  FORTRAN Subroutines 12-8
  Subroutine Format 12-9
    Introduction 12-9
    Parameter Passing Mechanism 12-9
    The Parameter Decoding Subroutine, F$RGMY 12-10
    Standard Assembly Language Subroutine Linkage Mechanism 12-10
    Multiple Subroutine Entry Points 12-13
  Subroutine Parameter Formats 12-14
    Introduction 12-14
    Basic Internal Data Formats 12-14
    Relationship Between Parameter Syntax And Parameter Format 12-17
  Passing String Values To and From Subroutines 12-19
    Conversion of Strings to Internal Form 12-19
    Conversion of Strings from Internal Form. 12-20

INTRODUCTION

Subroutines simplify the programming of repetitive tasks. A subroutine can take the place of a program section that is identically used at several points in a program. Program and storage space are conserved by changing a repetitive section of code with a single subroutine.

In addition to subroutines that are contained within the body of a BASIC program, BASIC allows the use of external subroutines as well. An external subroutine is one that is external to the body of the program. Fluke BASIC allows use of external subroutines written in FORTRAN or Assembly Language.

OVERVIEW

This section describes the various methods of programming subroutines, both internal and external and the BASIC statements used for subroutine programming. This section also describes the recommended subroutine format for assembly language subroutines. This format will provide Assembly Language subroutines that are compatible with Fluke Enhanced BASIC. The information in this section also describes the formats of parameters which may be passed to subroutines and the relationship between the BASIC language CALL statement syntax and parameter formats. Appendix I describes the limitations for FORTRAN subroutines to be used with BASIC.

USING INTERNAL SUBROUTINES

The GOSUB statement provides an unconditional branch to another segment of the BASIC program. When program execution transfers to the line number specified by the GOSUB statement, BASIC remembers the line number of the GOSUB statement. When a RETURN statement is executed, program execution transfers to the next line following the GOSUB statement.

Refer to the GOSUB and RETURN statements for detailed information. GOSUB statements may also be part of ON-GOSUB and IF-THEN-ELSE statements.

Examples

The following example uses a subroutine to print a changing message. Other program examples may be found in the Reference volume under the following headings: GOSUB and GOTO.

```
10  !program example 1
20  A$ = "Subroutine Demonstration"
30  GOSUB 100
40  A$ = "This is a demonstration of a subroutine."
50  GOSUB 100
60  PRINT "The subroutine uses the field attributes"
70  A$ = " to make the example more interesting."
80  GOSUB 100
90  PRINT "All done."
95  END
100 ! reverse video display subroutine
110  PRINT CHR$(27) + "[7m" + A$ + CHR$(27) + "[m"
120  RETURN
```

USING EXTERNAL SUBROUTINES

Fluke BASIC programs may also use subroutines that are external to the body of the main program. These subroutines are in the form of machine-executable object code. Subroutines may be written in Fluke FORTRAN, TMS-99000 Assembly Language, or a user-supplied language that produces TMS-99000 Machine Language. Refer to the specific language manual for details.

There are several reasons for using external subroutines:

1. You may require the additional speed that may be provided by subroutines in a directly machine executable form.

2. You may want to use another program development system that produces TMS-99000 Machine Language.

3. You may be able to utilize existing libraries of FORTRAN or Assembly Language subroutines.

4. You may want to take advantage of your existing programming expertise in FORTRAN or Assembly Language.

The LINK, CALL, and UNLINK statements provide for using external subroutines.

The LINK statement loads external subroutines into the Instrument Controller memory during execution of a BASIC language program.

The CALL statement executes these subroutines and, if desired, exchanges parameters with the subroutines.

The UNLINK statement removes subroutines from memory.

See the LINK, CALL, and UNLINK statement descriptions for details.

Software Requirements

Additional software is required to create Assembly Language or FORTRAN subroutines for use with BASIC:

Assembly: The necessary Assembler, Linkage Editor, and other programs are contained in the 17XXA-201 Assembly Language Option.

FORTRAN: The Fluke FORTRAN Compiler, FORTRAN Interface Runtime Library, Object Translator program, and Linkage Editor programs are contained in the 17XXA-202 FORTRAN Language Option.

Subroutine Names

Subroutine names have from one to six characters.

The first character must be a letter; the other five characters can be any alphanumeric character(s).

The CALL statement differentiates the subroutine name from BASIC statements.

1. If the BASIC language keyword (verb) CALL precedes the subroutine name, there are no further restrictions on the subroutine name.

2. If CALL does not precede the subroutine name, then the leading characters in the subroutine name must not conflict with a BASIC verb. The first of the following examples is a legal subroutine call, but the second example is not a legal subroutine because INIT is a valid BASIC verb.

```
EXAMPLE #1:     100 PINIT      !legal name
EXAMPLE #2:     100 INITP      !illegal name
```

All subroutine names and all identifiers of subroutines read by LINK have lowercase letters mapped into uppercase. For example, VADD and vadd refer to the same subroutine.

Assembly Language Subroutines

The Assembler program (17XXA-201 Assembly Language Option) creates relocatable object file subroutines (the files do not contain AORG [absolute origin] directives) that can be used with BASIC. The PARTIAL (partial link) command of the Linkage Editor program can be used to combine several relocatable object files into a partially linked module that can be used with BASIC. Any label in the Assembly Language subroutine which is also named in a DEF (define external symbol) assembler directive can be used as a subroutine entry point for a BASIC program if it also meets the subroutine name requirements described earlier.

Assembly Language Error Handler

For information on using the BASIC Error Handler along with Assembly Language Subroutines, see Appendix H.

FORTRAN Subroutines

FORTRAN programs normally run under control of the FORTRAN Runtime System program, which forms a link between the FORTRAN program and the Operating System program of the Instrument Controller. The FORTRAN Interface Runtime Library (FTN$IF.LIB) provides a link between the BASIC interpreter and FORTRAN subroutines used with BASIC. Whenever a FORTRAN subroutine is used with BASIC, this library must be linked with it. The standard FORTRAN libraries are not needed. Subroutine statements, format, restrictions and parameter passing are described in Appendix I.

After the FORTRAN subroutines have been written, it must be compiled, linked to the appropriate FORTRAN Library routines, and then processed by the Object Translator Program to produce a machine-language file that is compatible with the BASIC Interpreter program. This process is summarized below. See Appendix I for examples and the FORTRAN Compiler, Linker and Object Translator Program manuals for additional information.

1. Write and compile the FORTRAN subroutines.

2. Link the object file produced by the FORTRAN compiler with the FORTRAN Interface Runtime Libraries using the Linker program. Use the PARTIAL command to specify a partially linked version of the object file. The FORMAT command may specify either Compressed or ASCII format output.

3. Use the Object Translator Program to convert the object file produced by the Linker program to a form compatible with the BASIC Interpreter program.

Subroutine Format

Introduction

This information describes the recommended subroutine format. This format will provide Assembly Language subroutines that are compatible with Fluke BASIC and with future software products. The paragraph entitled Subroutine Parameter Formats describes the required format.

Parameter Passing Mechanism

When a CALL statement is executed, any parameter(s) included in the statement are passed to the specified Assembly Language subroutine.

- A CALL statement creates code (internal to the BASIC Interpreter) similar to the following Assembly Language statements:

```
blwp    @usrsub         ;transfer to user subroutine
data    n               ;number of parameters passed
data    p1              ;address of parameter 1
data    p2              ;address of parameter 2
...
...
data    pn              ;address of parameter n
```

- The parameter addresses (p1, p2, ... pn) can be either direct or indirect address pointers.

  1. If the low-order bit (the indirect bit) of the parameter address is zero, then the parameter is a direct pointer. The address passed to the subroutine is the address of the corresponding subroutine parameter.

  2. If the low-order bit of the parameter address is set to one, the parameter is an indirect pointer. The parameter passed to the subroutine is the address of a word that either contains the address of the parameter or contains the first indirect word in an indirect pointer chain.

     a. An indirect pointer chain is a series of indirect words. The first indirect word points to a second indirect word; the second indirect word points to yet another indirect word, etc.

b. Find the address of the parameter by tracing down the indirect pointer chain until a word having its lower order bit set to zero is found. This word contains the address of the desired parameter.

The following examples show the sequence of Assembly Language statements that are equivalent to the following two example CALL statements.

```
CALL STATEMENT:                    RESULTING STATEMENT SEQUENCE
1530 CALL VADD(A(), B(), K%)       blwp    @VADD
                                   data    3
                                   data    (address of A(0))
                                   data    (address of B(0))
                                   data    (address of K%)

1540 CALL REPORT                   blwp    @REPORT
                                   data    0
```

The Parameter Decoding Subroutine, F$RGMY

The parameter decoding subroutine, F$RGMY.OBJ, is part of the Instrument Controller Assembly Language Option. The F$RGMY subroutine should be linked with each object file which is to be linked with BASIC. Some of the actions of F$RGMY are described in the material that follows.

Standard Assembly Language Subroutine Linkage Mechanism

A listing that shows the standard subroutine linkage mechanism follows. An explanation of each numbered line follows the listing.

```
(1)             IDT   'USUB    '     ;name of subprogram
(2)             DEF   USUB           ;identifier of 'blwp' vector
(3)             REF   F$RGMY         ;name of linkage manager
(4)  USUB       DATA  WRKSPC,ENTRY   ;'blwp' vector to subroutine
(5)  ENTRY      BL    @F$RGMY        ;decode subroutine parameters
(6)             DATA  N              ;number of parameters expected
(7)             DATA  PARMS          ;pointer to parameter area
(8)  NAME       TEXT  'USUB    '     ;name of subprogram
                ...
                ...                  ;body of subprogram
                ...
(9)             RTWP                 ;return to BASIC Interpreter
(10) WRKSPC     BSS   32             ;subroutine workspace
(11)            DATA  NAME           ;pointer to subroutine name
(12)            DATA  START          ;pointer entry address
(13)            DATA  1              ;subroutine type flag
(14) PARMS      BSS   2*N            ;parameter address area
                ...
                ...                  ;other subroutine data
                ...
                END
```

Numbered line explanation:

(1) The IDT statement provides a module name for the Instrument Controller Linkage Editor program.

- The module name field must be enclosed by apostrophes (single quotation marks).
- The module name field is one to eight characters long.

(2) The DEF statement makes the entry name (i.e., address of transfer vector) available to BASIC and to other subroutines that can call this subroutine.

(3) The REF statement declares the routine F$RGMY as an external name (i.e., defined in another object program file).

(4) This DATA statement is the transfer vector used by BASIC to transfer control to the subroutine.

- The first word assembled for the DATA statement contains the address of the subroutine workspace.
- The second word assembled for the DATA statement contains the entry address of the subroutine.

(5) This is the first instruction executed by the subroutine. The routine F$RGMY, supplied with the Assembly Language Option, should be linked with each object (.OBJ) file to be linked with BASIC. The actions of F$RGMY are described as each action is encountered.

(6) This DATA statement is a parameter to F$RGMY. The statement gives the number of parameters expected by the subroutine. If the number of parameters passed to the subroutine does not match the number expected, F$RGMY displays error 706 (parameter count mismatch).

(7) This DATA statement, another parameter to F$RGMY, tells F$RGMY where to place the subroutine parameter addresses.

- F$RGMY converts all parameters with indirect bits set into ordinary parameters. That is, F$RGMY traces down an indirect pointer chain to find the direct pointer for the parameter.

- The parameter area, starting at PARMS, is filled with parameter pointers so that:

```
PARMS   DATA    (address of parameter 1)
        DATA    (address of parameter 2)
        ...
        DATA    (address of parameter n)
```

- If the number of parameters is less than or equal to eight, R0 through R7 can be used to hold the parameter addresses generated by F$RGMY. In this case, statement number (7) should be:

```
DATA    WRKSPC
```

- If no parameters are passed to the subroutine (i.e., n = 0), omit DATA statement number (7).

(8) The TEXT directive gives the name of the subroutine to F$RGMY.

- The subroutine name field must be enclosed by apostrophes (single quotation marks).

- The subroutine name field is six characters long. If the name is less than six characters, start the name at the left character position and enter blank spaces into the remaining character positions.

- The TEXT directive provides information about the sequence of subroutine calls to be used in future error handling capability.

(9) The RTWP instruction returns control to the BASIC Interpreter or to the module that called this subroutine. Note that F$RGMY sets the subroutine return address to point past the DATA statements that make up the caller's parameter list.

(10)The BSS statement reserves memory for the subroutine workspace.

NOTE

The placement of statements (11), (12), and (13) is critical. They must follow the subroutine workspace.

(11)This DATA word holds the address of the subroutine name defined by statement number (8). This word is set by F$RGMY.

(12)This DATA word holds the subroutine entry address [i.e., address of statement number (5)]. This word is set by F$RGMY.

(13)This DATA word is a subroutine type flag.

- This word should contain the number 1 (in binary).
- This word is not currently used by F$RGMY but is defined for future product compatibility.

(14)The BSS statement reserves space for the subroutine parameters.

- Omit this area if no parameter are passed.
- See statement number (7) for an explanation of the contents following a call to F$RGMY.

Multiple Subroutine Entry Points

The use of alternative workspaces and parameter areas is at the programmer's discretion. If a subroutine is entered at more than one point, statements (2) and (4) through (8) of the Standard Subroutine Linkage Mechanism should be present for each entry.

Subroutine Parameter Formats

Introduction

The information here describes the formats of parameters which may be passed to Assembly Language programs and FORTRAN programs and the relationship between the BASIC language CALL statement syntax and parameter formats.

Basic Internal Data Formats

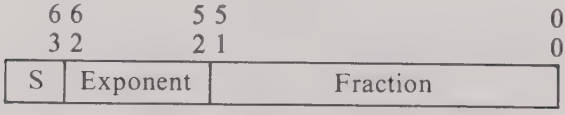
Parameters passed to Assembly Language or FORTRAN subroutines can be in one of three formats: Integer, Floating Point, or Character String.

- In the Integer format, parameters are 16-bit binary words arranged in the following sequence:

1. The word is in twos-complement form.

2. Integers are always aligned on an even (word) address.

- In the floating-point format, parameters are 63-bit words arranged in the IEEE double-precision format:



Where: S is the sign bit (1 = negative)
Exponent is a biased (bias = 1023) binary exponent
Fraction is a binary fraction of the form 1. Fraction (The 1 bit to the left of the fraction is implied. The binary point is implied to be left of bit position 51.)

1. This means that a (normalized) floating-point number has the value:

$$f = (-1)^{s} * 2^{(\text{Exponent} - 1023)} * (1.\text{Fraction})$$

2. All floating-point values used as parameters are normalized and are represented as 4-word values in memory as shown below.

3. Floating-point numbers are aligned on even (word) addresses.

Parameter pointer ⟶	Sign, exponent, hi-order fraction
+ 2 bytes	Next fraction word (bits 47-32)
+ 4 bytes	Next fraction word (bits 31-16)
+ 6 bytes	Last fraction word (bits 15-0)

- In the Character-String format, the parameter is the address of the string-head portion of a character string. A character string consists of a string-head and a list of singly linked 16-character string blocks.

1. String-head format is shown in the following figure:

Parameter pointer ⟶	Address of the block
+2 bytes	Number of characters in the string

a. The first word is a pointer to the first string block.

b. The second word is an integer that gives the number of characters in the string.

2. The string-block format is shown in the following figure:

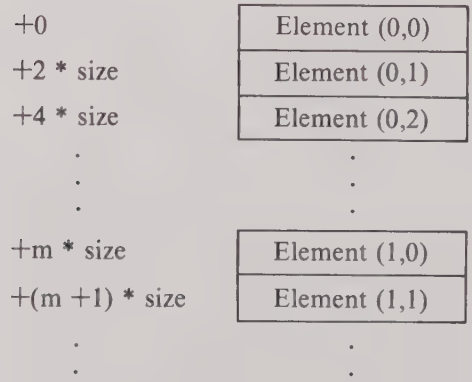
+0 bytes	Address of the next string block	
+2 bytes	Character 1	Character 0
+4 bytes	Character 3	Character 2
.	.	.
.	.	.
.	.	.
+16 bytes	Character 15	Character 14

a. Each string-block has 16 characters.

b. The characters in each string block are in reverse order (character 2 has an address 1 byte greater than character 3, for example) due to the internal organization of the BASIC Interpreter.

c. The last string block in the character string must have 0 set into the location for the address of the next string block because zero is never a valid address for a string block.

Arrays of scalar data types are allocated in row-major order (i.e., so that the subscript furthest to the right varies most quickly as the array elements are accessed in storage order). Arrays of integer and floating-point numbers are simply iterated forms of the representations described earlier. As the following figure shows, character-string arrays consist of a series of string-head cells.



Relationship Between Parameter Syntax And Parameter Format

Each parameter passed to an Assembly Language or FORTRAN subroutine is an address (i.e., parameters are passed by reference). The address can be the address of a variable, the address of an array, or the address of a temporary location (generated to hold the result of an expression). The BASIC Interpreter evaluates parameters according to their syntax type. The parameter syntax can be a variable identifier, array name, array element, expression, or constant.

- If a parameter is a variable identifier, the BASIC Interpreter uses the address of the variable as a parameter.

  1. The subroutine can return a result by changing the value of the named identifier.

  2. The "a%" in the following statement is an example of the variable identifier parameter syntax.

```
100 glorp(a%)
```

- If the parameter is an array name, the BASIC Interpreter uses the address of array element (0) or (0,0) as the parameter. In this way, an entire set of values may be made available to a subroutine.

  1. An array name is the array identifier followed by the characters ().

  2. An example of the array name parameter syntax is a$() in the following example:

```
110 redo(a$())
```

  3. The array name supplies no information about the array size. If the array size is to be made available to the subroutine, the size must be another parameter. For example, the 10% in the statement:

```
110 redo(a$(), 10%)
```

  4. A virtual array cannot be used as a parameter in this fashion because it exists in a file and thus has no fixed memory address.

- A single array element is copied to a temporary location. The address of this temporary location is used as the parameter. The "a%(i% * 5%)" in the following statement is an example of the array-element parameter syntax.

```
140 saddle(a%(i% * 5%))
```

- An expression, in this context, consists of at least one unary or binary operator or a function call. The value computed by the expression is placed in a temporary location. The address of this temporary location is used as the parameter.

  1. The parameter format (Integer, Floating Point, or Character String) is determined by the normal rules of BASIC expression evaluation.

  2. The "-sin(a) * fna(int(a))" in the following statement is an example of the expression parameter syntax.

```
120 undo(-sin(a) * fna(int(a)))
```

- A constant is copied into a temporary location. The address of this temporary location is used as the parameter. An example of a constant parameter syntax is the "abc" string in the following statement:

```
130 garble("abc")
```

Passing String Values To and From Subroutines

The BASIC Interpreter uses a linked-list data structure to provide variable-length strings. This internal data format is often difficult for Assembly Language programmers to use. The 1722A BASIC Interpreter provides two string conversion functions, which are called by the TMS-99000 XOP instruction, to convert to and from the BASIC internal string formats.

NOTE

It is absolutely essential that Assembly Language programs not change either the string head pointers or the link fields in BASIC strings. The penalty for not observing these rules is system failure. It is recommended that only the following interface functions be used for string parameter manipulation.

Conversion of Strings to Internal Form

In order to return a string result an Assembly Language program must use the string replacement routine of BASIC. The interface of this function strongly resembles that of the string unpacking routine.

1. The address of the string head (which is passed as a parameter in the BASIC CALL statement) is placed in the Assembly Language program's R0.

2. The address of a buffer (allocated by the Assembly Language program) is placed in the Assembly Language program's R1.

3. The length of the string contained in the buffer, whose address is in R1, is placed in the Assembly Language program's R2.

After these parameters have been set by the Assembly Language program the following instruction should be executed:

```
xop     r1,1
```

This will cause a transfer of control to the BASIC Interpreter string allocation routine. When the BASIC string allocation routine returns no changes will have occurred in any of the Assembly Language program's registers or string buffers.

The following program fragment should serve as an example the use of this interface function:

```
        mov     @strprm,r0      ;r0 = ptr to string parameter
        li      r1,buffer       ;r1 = ptr to buffer with new ...
*                               ;... string contents
        mov     @length,r2      ;r2 = length of new string
        xop     r1,1            ;call BASIC replacement function
        ...
buffer  bss     MAXLNG          ;ASCII string buffer
length  data    0               ;length of string in "buffer"
```

Conversion of Strings from Internal Form

Conversion of a string from BASIC's internal (linked-list) form into a packed ASCII buffer is done as follows:

1. The address of the string head (which is passed as a parameter in the BASIC CALL statement) is placed in the Assembly Language program's R0.

2. The address of a buffer (allocated by the Assembly Language program) is placed in the Assembly Language program's R1.

3. The length of the buffer, whose address is in R1, is placed in the Assembly Language program's R2.

4. An offset is placed in the Assembly Language program's R3. This offset tells the BASIC Interpreter how many initial characters to skip in the string before storing characters in the Assembly Language program's buffer. An offset of zero will retrieve characters starting with the first character in the string (that is, zero characters at the start of the string are skipped).

After these parameters have been set by the Assembly Language program the following instruction should be executed:

```
xop     r0,1
```

This will cause a transfer of control to the BASIC Interpreter string unpacking routine.

The data returned by the BASIC Interpreter will tell the Assembly Language program about the string actually placed in the buffer:

1. Characters from the string will be placed in the buffer pointed to by the caller's R1.

2. R2 will be set to contain the actual number of characters placed in the buffer pointed to by R1. This value will be between zero and the initial value of R2 (the maximum buffer length).

3. The processor's CARRY flag will be reset if the string returned was not longer than the buffer. If the CARRY flag is set (string longer than buffer), the value of R2 will be equal to its initial value; the ASCII buffer will be filled with the number of characters indicated by R2.

4. No changes will be made to the contents of R0, R1, or R3.

As an example:

```
        mov     @strprm,r0      ;r0 = ptr to string parameter
        li      r1,buffer       ;r1 = ptr to string buffer
        li      r2,MAXLNG       ;r2 = size of string buffer
        clr     r3              ;r3 = 0 offset
*                               ;(retrieve all chars in string)
        xop     r0,1            ;call BASIC unpacking routine
        jf      c               ;buffer overflowed
        seto    @ovrflg         ;signal buffer overflow
        endif                   ;
        ....
buffer  bss     MAXLNG          ;string buffer area
ovrflg  data    0               ;"buffer overflow" flag
```

Section 13
Program Chaining

CONTENTS

Introduction 13-3
Overview 13-3
Statement Definitions 13-4
  RUN Program Statement 13-4
  RUN WITH Statement 13-6
  EXEC Statement 13-6
  COM Statement 13-8
Virtual Arrays in Chained Programs 13-9
  Introduction to Virtual Array Chaining 13-9
  Example of Chaining a Virtual Array 13-10
  Using Sequential Data Files in Chained Programs 13-12

INTRODUCTION

Specific tasks to be executed sequentially may be handled by physically separate programs. Rather than use a GOSUB routine call, a RUN statement is used to call a separate program into main memory to be executed. This section describes the conventions and programming techniques for program chaining and for calling command files.

Program chaining is useful when the size of a program becomes larger than conveniently fits into user memory. Dividing a large program into a number of smaller logical segments allows more variable storage and processing capability in each program segment.

OVERVIEW

This section describes the RUN and CALL statements, virtual array files, and sequential data files in a chained program structure. Cross references are given for additional information available in this manual on the RUN statement, virtual arrays, and the OPEN and CLOSE statements.

STATEMENT DEFINITIONS

Program statements described below are used for chaining multiple programs that are stored on one of the file-structured devices. The Immediate Mode use of RUN is described in Section 2 of this manual.

RUN Program Statement

Usage: RUN [program name$]

The RUN statement restarts the program in main memory or loads and runs a BASIC program available in file storage.

- Without a file name, RUN will restart the program in main memory.
- A file name may be specified as a quoted string or as a string expression.
- The current FDOS command line is not changed.
- The program file will be searched for on the default System Device if the file name is not prefixed with a device name such as MFO:.
- If the file name extension is not .BAS or .BAL, it must be specified. Error 305 (file not found) results if the file is not found.
- When the file is located, it is loaded into main memory replacing the previous program, and control is transferred to it.
- Data stored in variables in the previous program is lost unless it was reserved in a common area with a COM statement or stored in a virtual array file.
- All files left open are available for use by the chained program. Virtual Array Dimension Statements (DIM#*n*) may be used to allocate virtual arrays (on the virtual array files left open) in the next program.

The following example searches for a program file named TEST2 on the System Device. If it is located, TEST2 is loaded into main memory and executed.

```
1050    RUN "TEST2"                    ! Chain to program TEST 2
1060    END
```

The following example searches for a program file under a name stored in string B$. If the file is located, it is loaded into main memory and executed.

```
1050    RUN B$      ! Chain to program named by B$
1060    END
```

RUN WITH Statement

Usage: RUN [program name$] [WITH command$]

The RUN WITH statement runs the BASIC program specified by the string "program name$". The optional WITH statement modifier causes the BASIC interpreter program to replace the FDOS command line with the string command$.

- This form of the RUN statement may be used to pass parameters to another BASIC program via the FDOS command line.
- Use the CMDLINE$ function to read the FDOS command line.

The RUN WITH statement is identical to the RUN statement with the following exceptions:

- The FDOS command line is changed to the string "program name$" followed by a space character and the string command$.

EXEC Statement

Usage: EXEC filename$ [WITH command$]

The EXEC statement permits a BASIC program to chain to a machine language program or to a command (".CMD") file.

- The EXEC statement is similar to the RUN statement (in that the program that executes the statement is terminated immediately).
- When the program specified by the EXEC statement terminates, control returns to FDOS unless the SET SHELL statement has been previously executed.
- The EXEC statement may be used in the immediate mode.
- Filename$ is the name of the the executable or command file to be executed.
- The optional WITH clause specifies a new command line to be passed to the new program.
- The string command$ will act as though it had been entered following the file name in a command to FDOS .

- If no device is specified, the device "SY0:" will be assumed.
- The only extensions permitted with this statement are ".CMD" and ".FD2". If no extension is specified, a file with the extension ".CMD" will be searched for first; if no ".CMD" file is found, the BASIC system will then look for a ".FD2" file with that name.
- The command line argument plus the length of the file name (less any ".CMD" or ".FD2" extension) may not exceed 80 characters. This limitation is imposed by FDOS. The actual command line passed to the EXEC'ed program will be file$ (without any device name or extension), and, if command$ is specified, a space and the string given as command$. The entire line is terminated by a carriage return character.

Example

BASIC STATEMENT	FDOS EQUIVALENT
`exec "edit" with "test.bas"`	`sy0:edit test.bas`
`exec "mf0:fup"`	`mf0:fup`
`exec "edit.fd2" with "foo"`	`sy0:edit.fd2 foo`

COM Statement

Usage: COM {variable list}

COM reserves variables and arrays in a common area for reference by chained programs.

- Only floating-point and integer variables may be stored in the common area.
- String variables may not be stored in the common area.
- String variables may be stored in virtual arrays for access by chained programs. This technique is discussed later in this section.
- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.

For example, assume that a chained program requires the use of three floating-point simple variables, an integer simple variable, a floating-point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM A, B, C, FX, D(24%), TX(100%)
        .
        .
        .
1050    RUN "B"
1060    END                              ! End of program A
```

The second program could then use:

```
10      ! Program B
20      COM L1, L2, L3, GX, K(24%), PX(100%)
        .
        .
        .
```

Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

VIRTUAL ARRAYS IN CHAINED PROGRAMS

Properly used, virtual arrays become a unique tool for controlling and keeping track of multiple task sequences. It is possible, for example, to structure a floppy disk that will continue with the next task to complete in a task list, even if the task is carried to another Controller and given a RESTART. The paragraphs that follow discuss these techniques.

Introduction to Virtual Array Chaining

An advantage of using virtual arrays to pass chaining information is that even if a power failure occurs, the status of the processing performed by the set of programs is preserved in the virtual array. When the capability to survive power failures is not required, chaining information may be kept in a COM variable or array.

Virtual arrays can be used to control the execution of chained programs. When writing a set of chained programs, open a channel for a virtual array file with the number of elements matching the number of chained programs called from the main program. (to chain three programs, dimension the virtual array with 3 elements -- 0, 1, and 2.) Initialize this array to indicate no programs have been executed.

Each element of the array will take either an OFF or ON value which indicates whether or not its associated program has been executed. Each program sets an element of the array to the value which indicates that the program has been run. To begin, initialize the elements to 0 (zero) then set the elements to 1 (one) as each program is executed. If the programs are halted for any reason, the array values will show which program was interrupted.

In this way, a rerun of the main program would execute only those programs which have not yet been executed. Programs which were executed before the interruption will not be rerun or will be rerun only under the circumstances indicated by the programmer.

Virtual arrays may also be used as a common storage area for a set of chained programs without using the COM statement. Once a virtual array has been initialized, any program can use the array by using the proper open, dimension, and closing sequences as described in the section on Virtual Arrays.

Example of Chaining a Virtual Array

One program (named PROG0 in this example) executes a set of n programs in sequence so that if a program does not execute successfully, it may be reexecuted. The virtual array file in this example will be named CHAIN.BIN.

NOTE

Each of the programs (PROG1 through PROGn) can have a set of subordinate chained programs.

Assume there are five chained programs, with names PROG1, PROG2, PROG3, PROG4, and PROG5. An initialization program (called PROGIN in this example) is executed to create the file CHAIN.BIN and to initialize the chaining information required to indicate that none of the chained programs have been executed.

```
10      !
20      ! PROGIN - Initialize chaining information for PROG1
30      !
40      OPEN "CHAIN.BIN" AS NEW DIM FILE 1% SIZE 1%
50      DIM #1%, A%(5%)
60      FOR I% = 1% TO 5%
70      A%(I%) = 0%                     ! Set to not-run
80      NEXT I%
90      CLOSE 1%
100     RUN "PROG0"                      ! Start program execution
110     END                             ! Of PROGIN
```

The EXEC program below checks the elements of the array in the file CHAIN.BIN until a zero element is found:

```
10      !
20      ! PROG0 - CHAIN TO THE NEXT INCOMPLETE PROGRAM
30      !
40      !
50      OPEN "CHAIN.BIN" AS DIM FILE 1%
60      DIM #1%, A%(5%)
70      FOR I% = 1% TO 5%
80      IF A%(I%) <> 0% THEN 120
90      ! FOUND A PROGRAM THAT NEEDS TO BE RUN
100     CLOSE 1%                        ! Release Channel 1
110     RUN "PROG" + NUM$(I%, "#") ! Chain to PROG 1
120     NEXT I%
130     !
140     ! If control reaches this point,
150     ! all programs have been completed.
160     !
170     CLOSE 1%
180     KILL "CHAIN.BIN"                ! Release file space
190     PRINT "ALL PROGRAMS COMPLETED."
200     END                             ! Of EXEC
```

Each of the programs, PROG1 through PROG5, performs the processing required of them, but each should follow this format (PROG1 is illustrated).

```
10      !
20      ! PROG 1 - Explanation of total program function
30      !
        .
        .
        .
10000   OPEN "CHAIN.BIN" AS DIM FILE 1%
10010   DIM #1%, CH%(5%)
10020   CH%(1%) = 1%                    ! Signal that PROG1 is complete
10030   CLOSE 1%                        ! Rewrite status to file
10040   RUN "PROGO"                     ! Chain through "EXEC"
10050   END                             ! Of PROG1
```

Using Sequential Data Files in Chained Programs

Sequential data files may be used as common data storage areas for chained programs. The major difference between using the sequential file as opposed to the virtual array file is the difference in access methods. Sequential files are non-random and must be read from or written to sequentially. They are not array oriented and, therefore, cannot be referenced via a dimensioned array name and element. If it is convenient to use all data in a file in the order it has been stored, a sequential data file is easier to use than a virtual data file.

For example, an instrument taking multiple readings produces data (e.g., voltages) that should be stored for processing by a separate program module. That data may be written into a sequential file by the data collection segment of the program system (assuming use of chained program techniques) in the order the data was read by the instrument. A report generating program segment may then process the data.

The principal use of sequential data files is the storage of variable length ASCII character strings. Sequential files should not be used for the storage of arbitrary binary data since some characters (CTRL/Z, Carriage Return, Line Feed) have a special meaning to the BASIC system's input and output routines.

ASCII data may be written to a sequential file or to an RS-232-C port by means of the PRINT statement. For example:

```
1030 PRINT #5%, "VALUE"; I%; ","; J%
```

Similar data may be read by an INPUT statement:

```
4010 INPUT #3%, A%(0%..5%)
```

Note also that the contents of a sequential file may be examined or sent to a printer by the File Utility Program (FUP). This is not true of virtual array files.

Section 14
Touch Sensitive Display

CONTENTS

Description 14-3
Introduction 14-3
Using the Display for Output 14-4
The ASCII Character Set 14-5
Alternate Character Sets 14-6
Display Output Statements 14-7
PRINT Statement 14-7
PRINT USING Statement 14-8
CHR$() String Function 14-9
TAB String Function 14-9
CPOS String Function 14-10
Special Display Control Characters 14-12
Display Control Sequences 14-13
ANSI Compatible Display Control Sequences 14-14
Display Control Character Sequences 14-16
Cursor Positoning and Display Scrolling 14-17
Cursor Movement 14-17
Display Scrolling 14-18
Erasing 14-18
Mode Commands 14-19
Character Visual Attributes 14-20
Field Attributes 14-22
Character Attributes 14-25
Non-destructive Display Character 14-26
Character Size 14-27
Character Graphics 14-28
Keyboard Disable and Enable 14-32

Using the Display for Input 14-33
Display Input Statements 14-34
KEY Variable 14-34
ON KEY and OFF KEY Statements 14-35
WAIT FOR KEY Statement 14-36
An Interactive Display Program 14-39

DESCRIPTION

The Instrument Controller's display offers several useful features that enable the design of customized operator message outputs. These features include character graphics, display graphics, double-size characters, reverse video, double intensity, and blinking. In addition, the cursor control functions and erasing capabilities give the programmer complete control over the display.

Input from the operator of an instrumentation system controlled by a Fluke Instrument Controller is accomplished by means of the touch sensitive panel overlaying the display. This panel is capable of identifying 1 of 60 distinct blocks or touch areas when the operator applies moderate touch pressure. By integrating display messages with expected responses, an operator can be directed to supply needed numerical or decision inputs without interfering with the process at hand.

INTRODUCTION

This section discusses techniques for effectively using the Touch-Sensitive Display feature of the Instrument Controller. The discussion covers four general categories of display functions: output, controlling the display, mode commands, and input.

USING THE DISPLAY FOR OUTPUT

The usual use of the display is for output. The PRINT statement sends information to the display. The PRINT USING statement sends formatted information to the display. The information transferred to the display uses the characters found in the ASCII Standard Character set.

The ASCII Character Set

The Controller uses ASCII characters for display as shown in Appendix G. The first 32 characters in the ASCII Character Set (numbered 0 through 31) are defined as control codes, which include such functions as backspace (BS) or carriage return (CR). When generated by the keyboard, many of these control codes are intercepted, and a block (character number 127) is displayed in their place. However, the CHR$(n%) string function can be used within a program to generate any eight-bit code pattern for the display, including the control codes.

Many of the control codes reserved by the ASCII standard are not used by the Controller. However they are used by the display system to provide some useful symbols. These symbols are stored in a ROM and are separate from the graphics characters described later in this section. Like other displayable ASCII characters from the ROM, these symbols may be enhanced with higher intensity, reverse video, blinking, and underlining.

NOTE

The John Fluke Mfg. Co., Inc. reserves the right to make changes to the symbols contained in character generator ROMS.

Here is a short program that will display all of the displayable characters generated by the character ROM in double-size display format. The block (character number 127) is displayed in place of all control characters that are not displayable. The space character is just left of the "!". Touch the screen to clear the display.

```
10  ES$ = CHR$(27) + "[" \ BL$ = CHR$(127)
20  PRINT ES$ + "1p"; CPOS(2,5); BL$;
30  FOR I = 1 TO 6 \ PRINT CHR$(I); \ NEXT I
40  FOR I = 7 TO 13 \ PRINT BL$; \ NEXT I
50  FOR I = 14 TO 26 \ PRINT CHR$(I); \ NEXT I
60  PRINT BL$;
70  FOR I = 28 TO 31 \ PRINT CHR$(I); \ NEXT I
80  PRINT CPOS(4,5); \ FOR I = 32 TO 63 \ PRINT CHR$(I);\ NEXT I
90  PRINT CPOS(6,5); \ FOR I = 64 TO 95 \ PRINT CHR$(I);\ NEXT I
100 PRINT CPOS(8,5); \ FOR I = 96 TO 127 \ PRINT CHR$(I);\ NEXT I
110 WAIT FOR KEY \ PRINT ES$ + "p";
```

Alternate Character Sets

An alternate character set can be designed to allow mathematical or other special symbols, such as might be needed for foreign languages, to be displayed. Appendix E of the 1722A System Guide describes EPROM programming techniques to make use of this special ability. To select the alternate character set, send the control character SHIFT OUT [SO = CHR$(14)] to the display; to select the normal set, send SHIFT IN [SI = CHR$(15)] to the display.

Display Output Statements

The following paragraphs describe the statements used to transfer information to the display. Complete descriptions for each statement may be found in the Reference volume of this manual set.

PRINT Statement

Usage: PRINT [string1$] [; or ,] [string2$] [; or ,] [stringn$] [; or ,]

The PRINT statement is used in Fluke BASIC to send code information or characters to the display.

- Any number of print items (described above as [string1$] etc.) may follow the PRINT statement, up to the line length limit imposed by the Controller.
- All code sequences described in this section that affect the display must be transmitted via PRINT to the display.
- Character and code sequences for display purposes may be conveniently stored in virtual arrays and called when needed by a PRINT statement.
- An output channel (NEW) may be opened to the display (KB0:), using an OPEN statement. The PRINT statement would then direct display output to that channel number.

The following examples illustrate PRINT usage within the context of this definition:

STATEMENT	MEANING
`PRINT "Fluke Controller"`	Display "Fluke Controller"
`DIM A$(2)` `A$(0) = "Fluke Controller"`	Dimension a 3 string array. Place message in the first string.
`PRINT A$(0)`	Display message from the array element.
`OPEN "KB0:" AS NEW FILE 3`	Open channel 3 to the display.
`PRINT #3, "Fluke"`	Send message to the display channel.

PRINT USING Statement

Usage: PRINT USING {format description}, [string1$][, or ;][string2$]

The PRINT USING statement sends formatted data to the display. PRINT USING is identical to PRINT except for the use of the string mask for format description. The display format used is specified by a string mask in the format description portion of the statement. The string mask uses various ASCII characters to describe the appearance of the formatted output.

For example:

```
100 P = pi                      !an irrational number
105 PRINT P                     !print the full value
110 PRINT USING "#.##", P       !print the value to 2 places
```

causes the following display:

```
run
 3.141593
3.14

Ready
```

CHR$() String Function

Format: CHR$ (numeric expression)

The CHR$() function creates an eight-bit ASCII-coded string character from the lower eight bits of the integer value of the numeric expression.

The CHR$() function is most often used to send non-printable ASCII characters to the display, such as the ASCII "ESC" character used for display formatting. For example:

```
100 PRINT CHR$(27)+"[1p";"FLUKE"
```

displays "FLUKE" on the display in double size letters.

TAB String Function

Format: TAB (n%)

The TAB function creates a string of space characters that would move the current print position forward to column n+1. The value of n% must be between 0 and 80. TAB is intended for use with printers, but can also be used with the display.

The current print position is the total number of characters transmitted since the last carriage return character, including all non-printable or non-displayable characters.

- The current print position may be ahead of the display cursor if a display control character sequence was included since the last carriage return character. These character sequences are discussed later in this section.

- A Carriage Return and Line Feed character sequence will precede the string of spaces if the current print position is already at or beyond column n.

The following example illustrates an alternate method tab function for the display, using a defined function. The function works by returning the cursor to the beginning of the line and then moving right to the column specified by the function argument (column 33 in line 40). The string functions in line 30 change the function argument to string form and then remove space characters from either side of it.

```
10  ES$ = CHR$(27) + "["            ! Control sequence identifier
20  CR$ = CHR$(13)                  ! Carriage Return
30  DEF FNTB$(C) = CR$+ES$ + MID(NUM$(C),2,LEN(NUM$(C))-2) + "C"
40  PRINT FNTB$(33); "HERE"
```

CPOS String Function

Format: CPOS (line, column)

CPOS creates a string which, when sent to the display by a PRINT statement, positions the cursor at the specified line and column.

- The string is always eight characters long, in the form: ESCape [line% ; column% H. For example, CPOS(3,20) is equivalent to CHR$(27)+"[03;20H".
- The line and column are numeric expressions.
- If the expression for line or column is floating-point, it must be within the range of integers, and will be truncated to an integer.
- If either the line or column is less than zero, a value of zero is assigned.
- If the line or column is greater than 99, a value of 99 is assigned.
- Additional limits are imposed by the video display module:

  1. A line or column number of zero is interpreted as one.
  2. A line number greater than 16 is interpreted as 16.
  3. A column number greater than 80 is interpreted as 80.
  4. In double-size display mode, these limits are respectively line 1, line 8, and column 40.

The CPOS function may be assigned to a string variable or added to strings for display formatting. It may take various forms as shown in the following examples which display "Fluke Instrument Controller" at line 10, column 30.

Example 1:

```
10   A$ = CPOS(10,30)
20   B$ = A$ + "Fluke Instrument Controller"
30   PRINT B$
```

Example 2:

```
10   PRINT CPOS(10,30); "Fluke Instrument Controller"
```

Example 3:

```
10   PRINT CPOS(10,30) + "Fluke Instrument Controller"
```

The CPOS string function may also be used for more creative displays. The following example displays a scaled SIN function, using CPOS:

```
10 ! *** Display Sine Function ***
20 !
30 FOR X = 1 to 80              !Setup loop for display length
40   Y = 6 * SIN (2 * PI * (X/40)) + 9 !Compute sine function
50   PRINT CPOS(Y,X); '*';      !Position cursor and display *
60 NEXT X                       !Loop
```

Special Display Control Characters

Twelve of the ASCII control characters are used by the Controller's display module. These characters are listed in Table 14-1. The ASCII mnemonic is presented with the corresponding CHR$(n) for each of these characters.

Table 14-1. Special Display Control Characters

CHR$(N)	NMEMONIC	FUNCTION	RESULT
CHR$(7)	BEL	BELL	Activates the Instrument Controller Beeper.
CHR$(8)	BS	BACKSPACE	Moves the cursor to the left one column, if not already positioned at the leftmost column.
CHR$(9)	HT	HORIZONTAL TAB	Moves the cursor to the next tab stop, located every 8 columns.
CHR$(10) CHR$(11) CHR$(12)	LF VT FF	LINE FEED VERTICAL TAB FORM FEED	These three commands all move the cursor to the next lower line, in the same column. The display scrolls upward if the cursor is on the bottom line.
CHR$(14)	SO	SHIFT OUT	Moves the cursor to the beginning of the current line.
CHR$(15)	SI	SHIFT IN	Enables the alternate character set (if programmed in character generator EPROM).
CHR$(13)	CR	CARRIAGE RETURN	Enables the standard character set.
CHR$(24)	CAN	CANCEL	Cancels a display control character sequence if sent as part of the sequence.
CHR$(26)	SUB	SUBSTITUTE	Cancels a display control character sequence if sent as part of the sequence.
CHR$(27)	ESC	ESCAPE	Starts a display control character sequence, discussed later in this section.

DISPLAY CONTROL SEQUENCES

The Instrument Controller display uses two methods of defining sequences of ASCII characters for added control of the display. One of these methods is a subset of ANSI Standard X3.4. The other method is used for certain display controls. This method also provides compatibility with the mode control sequences used by the Fluke 1720A Instrument Controller.

These character sequences enable the user to customize operator message outputs. Among the capabilities provided are character graphics, double-size characters, reverse video, double intensity, and blinking. The cursor control functions and erasing capabilities give the programmer complete control over the display.

- Display control character sequences are sent to the display via a PRINT statement.
- CHR$(27) is used to generate the ESCape code that prefixes each control sequence.
- Use quotes around the characters that follow CHR$(27) so that PRINT will handle them as strings.
- These sequences are summarized in Appendix E.

ANSI Compatible Display Control Sequences

The ANSI compatible display control sequences are used to set or reset display modes. The generalized command format for ANSI compatible display control sequences is:

⟨esc[⟩ [param1] ; [param2] ; [paramn] term

where ⟨esc[⟩ is the sequence:

CHR$(27) + "["

and the parameters are defined as set or reset by "selective parameters" rather than only by numeric characters. These parameters are always a string of characters whose first character is a question mark (?), and the second parameter a numeric character between 1 and 8. Any number may be specified within a given command set, but ill-formed parameters are ignored.

- To SET a mode, terminate the escape sequence with a lower case letter 'h'.
- To RESET a mode, terminate with a lower case letter 'l'.
- An easy way to remember these two letters is:

  SET = logic 1 = high = "h"
  RESET = logic 0 = low = "l"

Table 14-2 summarizes the ANSI Standard Mode Selections; defaults are those listed as "Reset". Each of these items is described later in this section.

Table 14-2. ANSI Compatible Mode Selections

MODE	RESET (l)	SET (h)
?1	Field Attributes	Character Attributes
?2	Single Size	Double Size
?3	Disable Character Graphics	Enable Character Graphics
?4	Keyboard Unlocked	Keyboard Locked
?5	Opaque to Graphics	Transparent to Graphics
?6	Disable Character Display	Enable Character Display
?7	Disable Graphics Display	Enable Graphics Display
?8	Disable Cursor Display	Enable Cursor Display

For example, to disable the cursor display, execute the following statement in the immediate mode (or as part of a program):

```
PRINT CHR$(27) + "[?8l"
```

Display Control Character Sequences

Display control character sequences are used to control the positioning of the cursor, erasing the display, character enhancements (attributes), and 1720A compatible mode selection. The generalized command format for display control sequences is:

ESC [(task number) (;task number) (task letter)

- Use quotes around the characters that follow CHR$(27) so that PRINT will handle them as strings.
- The semicolon separator is used to separate multiple task numbers that have the same task letter.
- Use either + or ; to link strings together. For example, PRINT CHR$(27) + "[1;5;7m" selects high intensity, blinking, and reverse image for all characters that follow.
- Other characters for display may immediately follow a display control sequence. For example:

```
PRINT CHR$(27) + "[1;5;7m Fluke Instrument Controller".
```

Table 14-3 lists the 1720A compatible mode control sequences. Each of these is described in more detail later in this section.

Table 14-3. 1720A Compatible Mode Control Sequences

MODE	MODE CONTROL SEQUENCE
normal size	⟨ESC⟩+ "[p"
double size	⟨ESC⟩+ "[1p"
character graphics ON	⟨ESC⟩+ "[2p"
character graphics OFF	⟨ESC⟩+ "[3p"
enable keyboard	⟨ESC⟩+ "[4p"
disable keyboard	⟨ESC⟩+ "[5p"

Cursor Positioning and Display Scrolling

The following paragraphs describe the control character sequences required to control cursor movement, display scrolling, and cursor status.

Cursor Movement

The cursor control character sequences can move the cursor to any position on the screen, either relative to the current position or absolutely, by designating line and column. Table 14-4 presents the cursor controls, with the PRINT statement required to produce them for the display. The semicolon shown at the end of each PRINT statement inhibits the Carriage Return and Line Feed codes that normally follow a print statement. Note that the string function CPOS(L%,C%) produces an equivalent character string, and is used in an identical manner. The CPOS string function was discussed earlier in this section.

Table 14-4. Cursor Controls

ACTION	PRINT STATEMENT*	EXAMPLE
Up n lines	PRINT ES$ + "nA";	PRINT ES$ + "4A";
Down n lines	PRINT ES$ + "nB";	PRINT ES$ + "3B";
Right n columns	PRINT ES$ + "nC";	PRINT ES$ + "21C";
Left n columns	PRINT ES$ + "nD";	PRINT ES$ + "17D";
Direct to line, column	PRINT ES$ + "L,CH";	PRINT ES$ + "5,23H";
Direct to line, column	PRINT CPOS (L%,C%);	PRINT CPOS (5%,23%);

*This table assumes this previous assignment:
ES$ = CHR$(27) + "["

Display Scrolling

Scrolling commands allow movement of the entire display up or down when movement is beyond top or bottom limits. If the cursor is not at the top or bottom of the display, no scrolling will take place. Table 14-5 lists the scrolling commands, and the BASIC statement required to send them to the display.

Table 14-5. Scrolling Commands

ACTION	PRINT STATEMENT*	EXAMPLE
Scroll down one**	PRINT EX$ + "D";	PRINT EX$ + "D";
Scroll up one**	PRINT EX$ + "M";	PRINT EX$ + "M";
Next line scroll**	PRINT EX$ + "E";	PRINT EX$ + "E";

*This table assumes this previous assignment:
EX$ = CHR$(27)
**No scrolling takes place if cursor is not at screen top or bottom.

Erasing

Commands are provided to allow a program to erase all or part of a line or of an entire display. Table 14-6 summarizes the erasing commands.

- Erase commands do not occupy a character display position.
- Partial erase commands are relative to the current cursor position.
- A semicolon placed at the end of an erase command will cause the cursor position to remain unchanged.

Table 14-6. Erase Commands

ACTION	PRINT STATEMENT
Erase to end of line	PRINT CHR$(27) + "[K";
Erase to end of line	PRINT CHR$(27) + "[0K";
Erase from start of line	PRINT CHR$(27) + "[1K";
Erase all of line	PRINT CHR$(27) + "[2K";
Erase to end of screen	PRINT CHR$(27) + "[J";
Erase to end of screen	PRINT CHR$(27) + "[0J";
Erase from start of screen	PRINT CHR$(27) + "[1J";
Erase all of screen	PRINT CHR$(27) + "[2J";

MODE COMMANDS

Mode commands affect eight areas: the visual attributes of displayed characters, the character size, the alternate graphics characters, the disabling of keyboard inputs, the interaction of the character plane with the graphics plane, the enabling of the character and/or display graphics displays, and the suppression of the cursor display.

- Mode commands do not occupy a display character position.
- Each of these areas is discussed separately below.
- 1720A compatible mode commands are noted individually.

Character Visual Attributes

The visual attributes of a character (blinking, underlined, reverse video, and highlighted) may be changed by a BASIC program. Two methods of modifying visual attributes are used: field attributes and character attributes.

- Visual attributes are modified by sending a visual attribute command string or enhancement string to the display via a PRINT statement.
- Field attributes operate on an entire field of the Controller's display; they are in effect until cancelled, overwritten by another enhancement string or overwritten by any ASCII character.
- When field attributes are used, a single or multiple enhancement command occupies one character position on the display, regardless of the length of the character string required.
- Character attributes operate on a character-by-character basis. Once an enhancement string is sent to the display, each succeeding character sent to the display takes on those attributes. These attributes may be cancelled or modified on a character-by-character basis.
- Character attributes are transparent; they require no display space.
- Four different enhancements are available: high intensity, underlining, blinking, and reverse image (dark characters on light background).
- Enhancement commands may be used in multiple combinations.
- The Instrument Controller will also respond to the display control commands used by the Fluke 1720A Instrument Controller. The Instrument Controller must have the field attribute mode set in order to use the 1720A display control commands.

Table 14-7 presents the enhancement commands, with the PRINT statement required to produce them for the display. The semicolon shown at the end of each PRINT statement inhibits the Carriage Return and Line Feed codes that normally follow a print statement. The enhancement commands are the same, regardless of which attribute mode is set.

Table 14-7. Character Enhancement Commands

COMMAND	PRINT STATEMENT
Enhancements OFF	PRINT ES$ + "m";
Enhancements OFF	PRINT ES$ + "0m";
High Intensity	PRINT ES$ + "1m";
Underline	PRINT ES$ + "4m";
Blinking	PRINT ES$ + "5m";
Reverse Image	PRINT ES$ + "7m";

NOTE: This table assumes the previous assignment:
ES$ = CHR$ + "["

Field Attributes

Field attributes affect the entire display of the Controller from the point where they are placed on the screen, until cancelled. Field attribute mode is the default display mode after the Controller is restarted, or cold-started. Field attribute mode may also be set by sending the string:

```
CHR$(27)+"[?11"
```

to the display via a PRINT statement.

- Overwriting a display location containing an enhancement command will delete the enhancement.
- Each enhancement command cancels all previous enhancements.
- Each field attribute command requires one character location on the display when displayed, regardless of the length of the character string required. This can cause unexpected results when designing highly formatted screens.

When field attributes are used, the underline and reverse image enhancements are subject to some limitations imposed by the video module. The refresh scanning rate of the display exceeds the rate at which characters are written into the display memory. As a result, these character enhancements will momentarily cause the entire remaining display to have underlines or light background until the enhancements OFF command is written into the display.

- This limitation only applies when using field attributes when sending characters to the display.
- The character attribute mode of the video module is not subject to this limitation.
- Programs that use 1720A type display enhancements are compatible in the field attribute mode.

To avoid this limitation, place the enhancements OFF command one character location past the future location of the last character of the message. Then back up to one location ahead of the start of the message and print the enhancement selections along with the message. Using the Display Worksheet (Fluke Part Number 533547, pad of 50) makes this task easier. Remember that each enhancement command occupies one display location and that the cursor moves forward one location after a command is placed.

The following example shows the correct sequence of statements to display "FLUKE" in blinking letters on line 4, column 14. A ruler-line is displayed on line 3 for a column number reference. When the program is run, notice that the string "FLUKE" appears at line 4, column 14 because the display enhancement character sequence sent in line 60 takes up one display location.

```
10    ES$ = CHR$(27) + '['       ! set up escape sequence
15    PRINT ES$+'?11'            ! field attributes
20    PRINT ES$; '2J'            ! erase display
25    PRINT CPOS(3,1);"12345678901234567890" !ruler for reference
30    PRINT CPOS(4,19);          ! move to clear end of message
40    PRINT ES$; 'm'             ! disable all enhancements
50    PRINT CPOS(4,13);          ! move to beginning of message
60    PRINT ES$; '5m';           ! enable blinking letters
70    PRINT 'FLUKE'              ! display message
```

The following example displays "Fluke" as in the last example. This time, the test string is stored in a string variable in line 20. The length of the string is checked in line 30 and then used in line 70 to position the disable enhancements command at the proper place in the display. Line 100 enables all enhancements in a single command that will occupy one display location.

```
10 ES$ = CHR$(27) + "["   ! standard escape sequence
20 M$ = "Fluke"           ! put test string here
30 LX = LEN(M$)           ! now store length in LX
40 PRINT ES$+"2J"         ! clear screen
50 PRINT ES$+"?11"        ! field attributes
60 PRINT CPOS(3,1);"12345678901234567890" !ruler for reference
70 PRINT CPOS(4,13 + LX + 1); !cursor to end of string
80 PRINT ES$;'m';         ! disable enhancements
90 PRINT CPOS(4,13);      ! cursor to beginning of string
100 PRINT ES$+'1;4;5;7m'; ! send the attributes out
110 PRINT M$;             ! print test string
120 END
```

Notice that in both of the preceeding programs, the CPOS() function is used to place the current print location at row 4, column 13 (line 50 in example #1, line 90 in example #2). Since the enhancement string requires one display location, the message appears at line 4, column 14.

To experiment with the effects of field attributes, try removing attributes from line 100, one at a time, to see their effects on the display. Next, change line 70 to read:

```
70 PRINT CPOS(4,13 + LX%); !cursor to end of string
```

Note the effect of this change on the balance of the display. Now add this line:

```
115 PRINT CPOS(10,13);'FLUKE INSTRUMENT CONTROLLER'
```

Note that the attributes stay set, because last character in the string "FLUKE" overwrites the disable enhancements command which line 70 placed at row 4, column 18.

Character Attributes

Character attributes achieve the same net effect as field attributes. Character attributes use the same character enhancement commands as field attributes, however, the Instrument Controller must first be set to the character attribute mode by sending the following string to the display:

```
CHR$(27)+'[?1h'
```

When character attributes are enabled, the Instrument Controller's display responds to the character sequences described in ANSI Standard X3.4. This standard defines sequences of ASCII characters used to control the visual characteristics of a video display module.

- The character sequences used for display enhancements are listed in Table 14-7, which was presented earlier in this section.
- Character attributes are said to be "transparent" and do not require space on the display screen.
- After setting the character attribute mode as described above, send the appropriate character sequences to the display module (using the PRINT statement) to modify the visual characteristics of the information displayed on the screen.
- Once a display enhancement character sequence is sent to the display, all subsequent characters sent to the display acquire those visual characteristics. This remains true until another display enhancement character sequence is sent to the display.

The following program illustrates the use of character attributes. When the program is run, the test string "FLUKE" is displayed in flashing characters. Notice that the test string appears at line 4, column 13 since the display enhancement character sequence sent in line 60 does not require any display space. Compare this program with the program presented earlier in this section under the heading: Field Attributes.

```
10 ES$ = CHR$(27) + "[" !predefine esc [ sequence
20 M$ = "FLUKE"         !test string
30 L% = LEN(M$)         !length of test string
40 PRINT ES$+"2J"       !clear screen
50 PRINT ES$+"?1h"      !character attributes
60 PRINT ES$+"5m"       !blinking attribute
70 PRINT CPOS(4,13);M$  !print the test string @ 4,13
80 PRINT ES$+"m"        !reset attributes
90 END
```

Non-destructive Display Character

The non-destructive character is used to modify tne visual attributes of characters that are already on the display screen. Thus, a program could display a list of choices (a menu), then after allowing the user to select an option, display that option by modifing its visual attributes.

The non-destructive character is

CHR$(27)+ "="

- The non-destructive display character is unique because it may be sent to the display, *overlaying* an existing character, without destroying (erasing) that character.
- Any character that is overwritten with the non-destructive character takes on the visual attributes in effect at that time. In essence, the new attributes for a given character or characters are "painted" over them using the non-destructive display character.
- The non-destructive character advances the cursor by one position.
- The non-destructive character is ignored when field attributes have been enabled.

The following program prints the test string "FLUKE" on the screen, waits, then causes the test string flash, character by character. Since character attributes do not require display space, the string remains intact and in the same screen position.

```
10 ES$ = CHR$(27) + "[" !predefine esc [ sequence
20 M$ = "FLUKE"          !test string
30 ND$ = CHR$(27)+"="    !non destructive character
40 L% = LEN(M$)          !length of test string
50 PRINT ES$+"2J"        !clear screen
60 PRINT ES$+"?1h"       !character attributes
70 PRINT CPOS(3,1);"12345678901234567890" !a ruler line
80 PRINT CPOS(4,13);M$   !print the test string @ 4,13
90 !send out the highlight string, then wait 1 second
100 PRINT ES$;"1m"; \ WAIT 1000
110 FOR I% = 0 TO L%     !use a loop to highlight the rest
115 !paint each one with the non destructive char
120 PRINT CPOS(4,13 + I%);ND$
130 WAIT 1000            !wait a second
140 NEXT I%              !do it again
150 PRINT ES$;'m';       !now reset everything
160 END
```

Character Size

Normal size: PRINT CHR$(27) + "[p";
PRINT CHR$(27) + "[0p";

Double size: PRINT CHR$(27) + "[1p";

Character size commands affect the entire display by changing the basic timing of the display scan. Double-size characters are doubled in both height and width, occupying the area of four normal-size characters. Such characters are more easily recognized from a distance.

- Normal size allows 16 lines of up to 80 characters each.
- Double-size allows 8 lines of up to 40 characters each.
- Character size commands do not occupy a display character position.
- Character size commands clear the screen and return the cursor to the home (upper left) position. If the command is not terminated with a semicolon, the cursor will then move down one line.
- The video display module will remain in normal- or double-size display mode, according to the most recent character size command, even if the program is completed.
- The video display module starts up in normal size display mode until commanded to double-size.
- The edit command cancels any previous double-size commands.

Character Graphics

Enable graphics characters: PRINT CHR$(27) + "[?3h"
Disable graphics characters: PRINT CHR$(27) + "[?3l"

The alternate graphics character set is presented in Figure 14-1. When graphics mode is enabled, these characters are displayed from alternate pattern generation circuitry in place of the numbers 0 through 9, and the : character.

- Don't confuse character graphics with display graphics. Display graphics is described in Appendix K of this manual.
- Graphics enable and disable commands do not occupy a character display location.
- Graphics characters cannot be given character enhancements.
- Graphics characters may be enabled and disabled as many times as desired within a single display. It is possible, for example, to display a box around a number.
- The names given to the graphics characters in Figure 14-1 are significant. If the characters are used as their name indicates, a display can be set up that changes in size only (not shape) as the display mode is changed between normal and double-size. Program code written in this manner can be used in other application programs.
- Graphics commands select a mode of the video display module. The module will remain in the mode most recently selected even if the program is completed.
- The video display module starts up in normal (graphics disabled) mode until commanded to enable graphics.
- Character graphics and the graphics plane can be used independently.

Figure 14-1. Alternate Graphics Character Set

CHARACTER	NORMAL SIZE	DOUBLE SIZE	FUNCTION
0			Top Right Corner
1			Top Left Corner
2			Bottom Right Corner
3			Bottom Left Corner
4			Top Intersect
5			Right Intersect
6			Left Intersect
7			Bottom Intersect
8			Horizontal Line
9			Vertical Line
:			Crossed Line

NOTES:

1. To enable Graphics Mode, send the display ESC [3p or ESC [?3h
2. To disable Graphics Mode, send the display ESC [2p or ESC [?3l
3. In Graphics Mode, characters in the left column are displayed as shown.
4. Use the character names as defined to construct illustrations that do not change form between normal and double size.

The following program displays the graphics characters, first in normal size, and then, when the screen is touched, in double-size. Touching the screen a second time will clear the display and reset it to normal size with character graphics disabled.

```
10   ES$ = CHR$(27) + "[" \ CL$ = ES$ + "2J"
20   PRINT CL$; ES$ + "3p"; CPOS(8,24); \ GOSUB 50
30   PRINT CL$; ES$ + "1;3p"; CPOS(4,5); \ GOSUB 50
40   PRINT ES$ + "2p"; CL$
45   STOP
50   PRINT "0 1 2 3 4 5 6 7 8 9 :"
60   WAIT FOR KEY \ K% = KEY \ RETURN
```

The following example program illustrates:

1. Use of the character graphics to draw a box.
2. Character size commands.
3. Display character enhancements.

This program will draw a box, and then display "FLUKE INSTRUMENT CONTROLLER" underlined and in reverse image within the box. Every two seconds the entire display will change from normal to double-size and then back. Touching the screen will reset all modes to normal and clear the screen.

In this example, the display is created by a subroutine. The same code is used for both normal and double-size. This is due to two separate techniques:

1. Graphics codes are used as defined by their names, without reference to how they look.
2. Cursor movement is relative to the center point, established before entering the subroutine.

```
10   ! Display Demonstration Program
20   !
30   ON KEY 0 GOTO 800              !Program exit path
40   !
50   ! String variables:
60   !
70   ES$ = CHR$(27) + "["           !Control sequence identifier
80   NS$ = ES$ + "p"                !Normal size
90   DS$ = ES$ + "1p"               !double-size
100  NC$ = CPOS(8,40)               !Normal size center
110  DC$ = CPOS(4,20)               !double-size center
120  GR$ = ES$ + "3p"               !Graphics mode
130  CG$ = ES$ + "2p"               !Clear graphics mode
140  D1$ = ES$ + "B" + ES$ + "D"  !Down one and left one
150  U1$ = ES$ + "A" + ES$ + "D"  !Up one and left one
160  L1$ = ES$ + "2D"               !Left two
170  HM$ = CPOS(1,1)                !Home position
180  !
190  ! First do it normal size:
200  !
210  PRINT NS$; NC$;                !Normal size, center.
220  GOSUB 390                      !Display
230  WAIT 2000                      !Wait 2 seconds
240  !
250  ! Then do it double-size:
260  !
270  PRINT DS$; DC$;                !double-size, center
280  GOSUB 390                      !Display
290  WAIT 2000                      !Wait 2 seconds
300  !
310  ! Then repeat the sequence:
320  !
330  GOTO 210
340  !
350  ! The display subroutine:
360  !
370  ! First draw a box:
380  !
390  PRINT GR$;                     !Graphics mode
400  PRINT ES$ + "14D"; ES$ + "2A"; !Left 14, up 2
410  !
420  PRINT "1";                     !Upper left corner
430  !
440  FOR I = 1 TO 28
450  PRINT "8";                     !Top line
460  NEXT I
470  !
480  PRINT "0";                     !Upper right corner
490  !
500  FOR I = 1 TO 5
510  PRINT D1$; "9";                !Right side
520  NEXT I
530  !
540  PRINT D1$; "2";                !Lower right corner
550  !
560  FOR I = 1 TO 28
570  PRINT L1$; "8";                !Bottom line
580  NEXT I
590  !
600  PRINT L1$; "3";                !Lower left corner
610  !
620  FOR I = 1 TO 5
630  PRINT U1$; "9";                !Left side
640  NEXT I
650  !
660  PRINT CG$                      !Clear graphics mode
670  !
680  ! Place the message in the box:
690  !
700  PRINT ES$ + "2B"; ES$ + "26C"; !Down 2, right past message
710  PRINT ES$ + "m";               !Display enhancements off
720  PRINT ES$ + "26D";             !Left, just before message
730  PRINT ES$ + "4;7m";            !Select character enhancements
740  PRINT "FLUKE  INSTRUMENT CONTROLLER";
750  PRINT HM$;                     !Cursor to home position
760  RETURN
770  !
780  ! Clean up the display before leaving:
790  !
800  K% = KEY                       !Empty the KEY buffer
810  PRINT CG$; NS$;                !Reset graphics, normal size
820  PRINT HM$; home position !normal size, home position
830  END
```

Keyboard Disable and Enable

Disable keyboard: PRINT CHR$(27) + "[?4h"
Enable keyboard: PRINT CHR$(27) + "[?4l"

When the keyboard is disabled, all programmer keyboard inputs except control codes are ignored.

- The Touch-Sensitive Display remains active for inputs.
- Disable and enable commands do not occupy a character display position.
- A semicolon (;) placed at the end of a keyboard enable or disable command inhibits the transmission of Carriage Return and Line Feed to the display.
- Keyboard disable/enable status is a mode of the display module. This status remains as defined in the most recent disable or enable command, even after a program ends.
- The Instrument Controller starts up with the keyboard enabled.
- The keyboard is reenabled by entering ⟨CTRL⟩/T.

USING THE DISPLAY FOR INPUT

The Touch-Sensitive Display of the Instrument Controller allows operation without a keyboard. This feature allows the user to design operator input prompts in application programs that relate directly to the task at hand. The operator is free to maintain his or her focus upon the assigned task without the distraction of a complex keyboard or non-relevant decisions.

Figure 14-2, Instrument Controller Programming Worksheet illustrates the relationship between the Touch-Sensitive Overlay or TSO and the locations of displayed characters. Note that the touch-sensitive area is divided into 60 blocks numbered sequentially from the upper left. Each block covers 3 double-size characters, or 12 normal-size characters. Columns and rows of both normal and double-size characters are numbered as the CPOS string function would address them. This grid pattern worksheet is a useful planning tool available from Fluke in pads of 50 sheets. Order Fluke part number 533547.

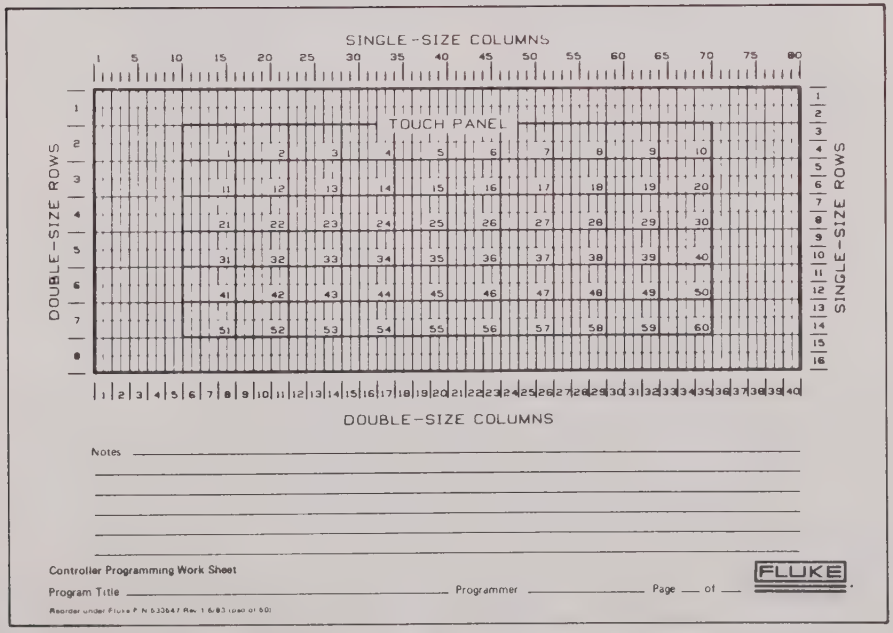


Figure 14-2. Controller Programming Work Sheet

Display Input Statements

The following paragraphs describe the KEY Variable, WAIT FOR KEY, ON KEY GOTO, and OFF KEY statements. Each of these BASIC statements is used by a BASIC program for getting user responses from the TSO.

KEY Variable

The system variable KEY contains the number of the last TSO block pressed. The KEY variable has the following characteristics:

- KEY is an integer ranging from 0 to 60.
- A KEY value of zero means that no TSO block has been pressed since the last time the value of KEY was used by a BASIC statement or since the last time a RUN command was executed.
- When KEY is accessed by a program (e.g., "K% = KEY" or "IF KEY = 0 THEN..."), it is set to zero.
- KEY may be used in any context that requires an integer variable.
- KEY cannot be assigned a value except by pressing the TSO.

ON KEY and OFF KEY Statements

Usage: ON KEY GOTO {line number}
OFF KEY

ON KEY GOTO (line number) is one form of the ON-event interrupt enabling statement described in the Interrupt Processing section.

- When the ON KEY statement is encountered, the number stored in KEY is repeatedly checked for non-zero.
- A zero value for KEY causes the GOTO to be ignored until KEY becomes non-zero.
- Control transfers to the line number following GOTO if a non-zero value for KEY is detected any time after the ON KEY Statement has been executed.
- The section of program referenced by GOTO may be an interrupt processing routine, with a RESUME statement, or it may be a program exit.
- After control has been transferred, the next RESUME statement transfers control back to the program line that would have been executed next if the non-zero KEY had not been detected.
- After control has been transferred, further KEY entries on the Touch-Sensitive Display are ignored until a RESUME statement is encountered.
- OFF KEY disables further checking of the value of KEY.

WAIT FOR KEY Statement

Usage: WAIT [time expression] FOR KEY

(time) FOR KEY is one form of the WAIT FOR (event) statement described in the Interrupt Processing section.

- When the WAIT FOR KEY statement is encountered, the number stored in KEY is checked for non-zero.
- The maximum time to wait may be specified by a number, numeric expression, or time expression.
- If the time to wait is expressed as a number or numeric expression, it must evaluate to an integer between –32768 and 32767.
- Time expressions allow the time to wait to be described in the folowing form:

  hh:mm:ss !hours:minutes:seconds

- A complete description of valid time expressions is described in the Reference volume of this manual set under: WAIT FOR.
- The statement is ignored if the specified time is zero or negative.
- The wait time is indefinite (until the Touch-Sensitive Display is pressed) if no time is specified.
- A zero value for KEY will cause program execution to stop until KEY becomes non-zero, or until the specified time has elapsed.
- A non-zero value for KEY will immediately terminate the wait condition, passing control to the next program statement.

The following example illustrates the KEY variable and the WAIT FOR KEY interrupt. Line 10 clears the KEY buffer. This resets any value it contained from touching the display before the program started. Line 20 halts the program until the Touch-Sensitive Display is touched. Line 30 prints the KEY number. Line 30 also clears the key buffer. If the buffer were not cleared, line 20 would detect it again, allowing line 30 to display the same key value repeatedly.

```
10   K% = KEY                          ! clear key buffer
15   PRINT "Touch me "                 ! display a message
20   WAIT FOR KEY                      ! enable key interrupt
30   PRINT 'KEY = '; KEY               ! display key value
40   GOTO 20
```

Sample displayed results:

```
KEY = 41
KEY = 1
KEY = 23
KEY = 57
KEY = 18
```

The following example program displays the number of each key in double-size directly under the spot that was touched.

- Flag KF% in line 80 enables the interrupt routine to clear the "Touch the Display" message from the display.
- After displaying a prompt message, line 120 halts the program until the display is touched. Then line 130 enables the KEY interrupt.
- Since the KEY buffer has a number in it from touching the display, line 130 immediately branches to the interrupt routine, disabling the KEY interrupt.
- Since the KEY flag KF% is initially zero, line 220 clears the "Touch the Display" message.
- Line 230 sets the KF% flag to one, so that subsequent passes through the routine will not clear the display.
- Lines 240 through 290 compute the position of the spot that was touched and display the KEY number at that spot.
- Line 300 reenables the KEY interrupt and branches back to the main routine. Since the ON KEY GOTO statement is still active, line 140 waits for another touch on the display.

- If a touch occurs within 10 seconds, the interrupt routine is reentered.
- If a touch does not occur within 10 seconds, line 150 disables the KEY interrupt and branches to line 80 starting the sequence over again.

```
10   ! *** Display Keys ***
20   !
30   ! Displays the KEY number of the spot touched. Clears
40   ! the display to start over if not touched within 10 seconds.
50   !
60   K% = KEY                  !Clear KEY buffer
70   ES$ = CHR$(27) + "["      !Display escape sequence
80   KF% = 0%                  !KEY flag
90   PRINT ES$; "2J"; ES$; "1p" !Clear display, double-size
100  PRINT CPOS(4,10);         !Position cursor
110  PRINT "Touch the Display!" !Display prompt message
120  WAIT FOR KEY              !Wait until display is touched
130  ON KEY GOTO 200           !Enable KEY interrupt
140  WAIT 10000                !Wait 10 seconds
150  OFF KEY                   !Disable KEY interrupt
160  GOTO 80                   !Loop
170  !
180  ! *** Key Interrupt Routine ***
190  !
200  K% = KEY                  !Get KEY number
210  IF KF% = 1% THEN 240      !First time through?
220  PRINT ES$; "2J"           !Clear screen
230  KF% = 1%                  !Set KEY flag
240  KI% = K% - 1%             !Compute KEY index
250  TR% = INT (KI% / 10%)     !Compute touch panel row
260  DR% = TR% + 2%            !Compute display row
270  DC% = 6% + (KI% - 10% * TR%) * 3% !Compute display column
280  PRINT CPOS(DR%,DC%);      !Position cursor to spot touched
290  PRINT USING "##", K%;     !Display key on spot touched
300  RESUME 140                !Wait for another key
```

An Interactive Display Program

The following program combines graphics and the KEY function to create an interactive display without a keyboard. The display is a matrix over the touch-sensitive area containing the characters 0-9, A-Z +, -, *, /, SPACE, DELETE, and ENTER. It requests the operator to enter information. As the operator touches the characters, they are entered into string OE$ and displayed. Previous characters are deleted when DELETE is touched. The operator entry subroutine returns to the main program (operator entry loop) when ENTER is touched. A typical user program could then use the operator entries in string OE$. This example program simply requests another entry.

The first section of the program (up to line 1620) sets up the display. Lines 1060 through 1460 create the display matrix by printing concatenated strings of graphic characters. Lines 1470 through 1620 print the characters within the matrix. The DUPL$() function is used in lines 1360 to 1450 to simplify drawing the boxes used at the bottom of the display.

The second section of the program requests operator entries. It compares the KEY value against limits and adds the corresponding ASCII character to string OE$.

```
1000 ! *** Interactive Display Program ***
1010 !
1020 ! Uses graphics and KEY function for operator keyboard
1030 !
1040 ! Set Up the Display:
1050 !
1060 DIM S$(11)                    !Dimension graphics variable
1070 ES$ = CHR$(27) + "["          !Display control identifier
1080 PRINT ES$ + "?3h"             !enable character graphics
1090 PRINT ES$ + "2J"              !clear the display
1100 PRINT ES$ + "?11"             !set field attributes mode
1110 T$ = CPOS(2,10) + "188888" !First line
1120 X = 9
1130 A$ = "488888"
1140 GOSUB 12100
1150 S$(0) = T$ + "0"
1160 T$ = "588888"                 !Part of 3rd, 5th, 7th lines
1170 A$ = ":88888"
1180 GOSUB 12100
1190 T$ = T$ + "6"
1200 FOR I = 2 TO 6 STEP 2   !Add CPOS to 3rd, 5th, 7th lines
1210 S$(I) = CPOS(I+2,10) + T$
1220 NEXT I
1230 T$ = "9"                      !Part of 2nd, 4th, 6th, 8th lines
1240 X = 10                        ! how many times
1250 A$ = DUPL$(" ",5)+"9"    !(5 SPACES, THEN 9)
1260 GOSUB 12100     !subr to make grfix chr string from a$
1270 S$ = T$
1279 ! Add CPOS to 2nd, 4th, 6th, 8th lines
1280 FOR I = 1 TO 7 STEP 2
1290 S$(I) = CPOS(I+2,10) + T$
1300 NEXT I
1310 T$ = CPOS(10,10) + "388888" !9th line
1320 X = 9
1330 A$ = "788888"
1340 GOSUB 12100
1350 S$(8) = T$ + "2"              ! 8's are 16 long
1360 S6$ = DUPL$(" ",6)            ! a string of 6 spaces
1370 SP$ = DUPL$(" ",16)           ! a string of 16 spaces
1380 T1$ = "1" + DUPL$("8",16) + "0"    ! a horizontal line
1389 ! set up box tops
1390 T$ = CPOS(12,10) + T1$ + S6$ + T1$
1400 S$(9) = T$ + S6$ + "1" + DUPL$("8",10) + "0" !10th line
1409 !set up box sides
1410 T$=CPOS(13,10) + "9" + SP$ + "9" + S6$ + "9" + SP$ + "9"
1420 S$(10) = T$ + S6$ + "9" + DUPL$(" ",10) + "9"
1430 T2$ = "3" + DUPL$("8",16) + "2"
1439 ! set up box bottoms
1440 T$ = CPOS(14,10) + T2$ + DUPL$(" ",6) + T2$ + DUPL$(" ",6)
1450 S$(11) = T$ + "3" + DUPL$("8",10) + "2"
1459 ! Display the matrix, disable character graphics
1460 PRINT S$(0..11); ES$ + "?31";
1470 A = 1                              !Display 1st row characters
1480 X = 48
1490 Y = 57
1500 GOSUB 10320
1510 X = 65                             !Display 2nd row characters
1520 Y = 74
1530 GOSUB 10320
1540 X = 75                             !Display 3rd row characters
1550 Y = 84
1560 GOSUB 10320
1570 X = 85                             !Display 4th row characters
1580 Y = 90
1590 GOSUB 10320
1600 PRINT CPOS(9,49)+"+";CPOS(9,55)+"- ";
1602 PRINT CPOS(9,61)+"*";CPOS(9,67)  +"/";
1609 ! label key boxes at bottom of screen
1610 PRINT CPOS(13,16)+"ENTER";CPOS(13,41)+"SPACE";
1612 PRINT CPOS(13,61)+"DELETE"
1620 PRINT ES$ + "?81"             !turn cursor off
1630 !
1640 ! ** Operator Entry Loop **
1650 !
1660 OP$ = "ENTER SOMETHING"      !Operator prompt
1670 GOSUB 10020                   !Call operator entry routine
1680 !
1690 ! User program to utilize operator inputs should go here.
1700 !
1710 GOTO 1660                     !Loop
1720 STOP                          !End of program
```

```
1730 !
10000 ! ** Subroutine to Accept Operator Entries **
10010 !
10020 OE$ = ""          !Null entry, display prompt & entry:
10039 !first time through, print the prompt, with a flashing '-'
10040 PRINT CPOS(1,1) + OP$ + ES$+"5m" + "-" + ES$+"m" + ES$+"K"
10050 GOTO 10080
10069 !this time don't make the '-' flash
10070 PRINT ES$+"m" + CPOS(1,1) + OP$ + " - " + ES$+"K" + OE$
10080 K = KEY                           !Empty KEY buffer
10090 WAIT FOR KEY                      !Wait for display touch
10100 K% = KEY                          !Get KEY number
10110 PRINT CHR$(7%);                   !Sound beeper
10120 !   Check for 0 - 9:
10130 IF K% < 11% THEN OE$ = OE$ + CHR$(47% + K%)
10140 !   Check for A - Z:
10150 IF K% > 10% AND K% < 37% THEN OE$ = OE$+CHR$(54% + K%)
10160 IF K% = 37% THEN OE$ = OE$ + "+" !Check for +
10170 IF K% = 38% THEN OE$ = OE$ + "-" !Check for -
10180 IF K% = 39% THEN OE$ = OE$ + "#" !Check for #
10190 IF K% = 40% THEN OE$ = OE$ + "/" !Check for /
10200 !   Check for Delete:
10210 IF K% = 59% OR K% = 60% THEN OE$=LEFT(OE$,LEN(OE$)- 1%)
10220 !   Check for Space:
10230 IF K% > 54% AND K% < 58% THEN OE$ = OE$ + " "
10240 !   Check for Enter:
10250 IF K% > 50% AND K% < 54% THEN 10270 ELSE 10070
10260 !   Display message:
10270 PRINT CPOS(16,1)+"Your last entry: "; ES$+"K";OE$;
10280 RETURN
10290 !
10300 ! ** Subroutine to Display Characters **
10310 !
10320 A = A + 2                         !Increment line number
10330 B = 13                            !Set column number
10340 FOR I = X TO Y                    !Loop on range of characters
10350 PRINT CPOS(A,B) + CHR$(I);        !Display character
10360 B = B + 6                         !Increment column
10370 NEXT I                            !Next character
10380 RETURN
10390 !
12000 ! ** Subroutine to Create Graphic Character Strings **
12010 !
12100 FOR I = 1 TO X                    !Loop on set of graphic parts
12110 T$ = T$ + A$                      !Add set to temporary string
12120 NEXT I                            !Next set
12130 RETURN
12140 !
32767 END
```

Section 15
Program Debugging

CONTENTS

Introduction 15-3
Overview 15-3
Debugging Tools 15-4
TRACE ON Statement 15-4
Line Number Tracing 15-5
Variable Tracing 15-8
Other Trace Options 15-11
TRACE OFF Statement 15-13
STOP ON Statement 15-13
STEP Command 15-14
CONT TO Command 15-15

INTRODUCTION

A program is seldom perfect the first time it is typed in. Errors are expected and Fluke BASIC provides several ways to detect and correct them. Programming errors are called bugs. The procedures used to find and correct errors are called debugging methods. This section describes these methods.

Errors fall into two categories: errors in syntax and errors in logic. Syntax errors are format or typographical errors, such as mismatched parentheses or an unrecognizable keyword. The BASIC interpreter will notify the programmer of most of these at the time the statement is first typed in and notify him of any remaining syntax errors the first time the program is run. The interpreter does not, however, notify the programmer of logic errors which show up as incorrect program results. Without additional tools these logic errors can be difficult to locate. This section presents the tools provided for locating logic errors.

OVERVIEW

This section describes debugging tools that are primarily useful for locating logic errors. These tools allow the programmer to observe program flow and variable assignment while the program executes statements. These statements do not check for syntax errors. They are designed to simplify the task of locating logic errors in the program.

DEBUGGING TOOLS

TRACE ON Statement

Usage: TRACE ON [#n,] [starting line number]
TRACE ON [#n,] [trace variable list]
TRACE ON [#n,] [line number] [trace variable list]

TRACE prints a record of line numbers encountered or changes in variable values.

- If a previously opened channel is specified, the results of the trace are sent to the channel. Otherwise the results are sent to the display.
- A starting line number for tracing may be specified. If it is not specified, tracing starts with the first line following the execution of the TRACE statement.
- Tracing is activated when the specified start line, or the first line, is encountered.
- TRACE may be used in either Immediate or Run Mode.

Line Number Tracing

A line number trace has the following forms:

STATEMENT	MEANING
TRACE ON	Trace line numbers from the first line and send the results to the display.
TRACE ON line number	Trace line numbers from the specified line and send the results to the display.
TRACE ON # channel	Trace line numbers from the first line and send the results to the open channel.
TRACE ON # channel, line number	Trace line numbers from the specified line and send the results to the display.

- A line number trace and a variable trace will not execute concurrently.
- A line number trace occurring after a variable trace specifies a new line number after which variable tracing will resume, provided no TRACE OFF occurred in the interim.

The following examples illustrate the results of different forms of line number trace statements.

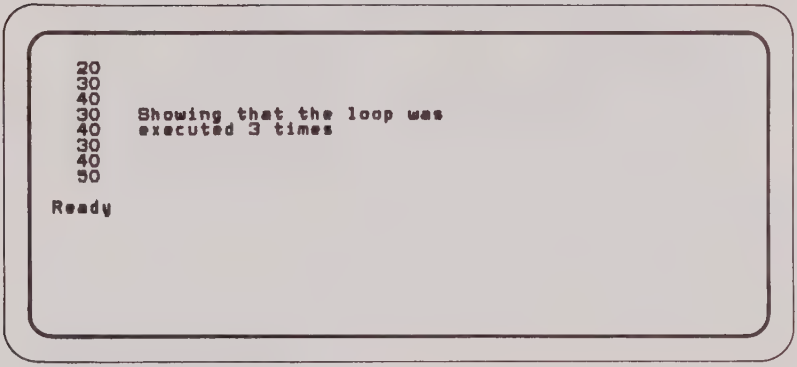
STATEMENT		RESULT
30	TRACE ON	Start a line number trace at the next line following line 30.
500	TRACE ON 1275	Start a line number trace when line 1275 is reached.
750	TRACE ON # 3%, 400	Start a line number trace when line 400 is reached. Send the trace output to channel 3.

The line number trace displays a series of numbers representing the line numbers or the statements executed. The following example illustrates typical results.

Program:

```
10      TRACE ON
20      I% = 0%
30      I% = I% + 1%
40      IF I% < 3% THEN 30
50      TRACE OFF
60      END
```

Results:


20
30
40
30 Showing that the loop was
40 executed 3 times
30
40
50
Ready

Variable Tracing

A variable trace has the following forms:

STATEMENT	MEANING
TRACE ON variable list.	Trace changes in value of selected variables from the first line and send the results to the display.
TRACE ON #channel, variable list	Trace changes in value of selected variables from the first line and send the results to the open channel.
TRACE ON line number, variable list	Trace changes in value of selected variables where the specified line is encountered and send the results to the display.
TRACE ON #channel, line number, variable list	Trace changes in value of selected variables from the specified line and send the results to the open channel.

- If a list of variables is specified, the trace is of changes in values of those variables. Otherwise, the trace is of line numbers encountered.
- A variable trace may specify one or more variables of any type: string, integer, and floating point.

- A variable trace of an array may use the form A() as the variable. The statement:

  TRACE ON A()

  means "TRACE ON all elements of array A".

- An array must be previously dimensioned before tracing.

- A variable trace and a line number trace will not execute simultaneously.

- Two or more variable traces will execute simultaneously. For example:

```
10 TRACE ON A
20 TRACE ON B
```

  is equivalent to TRACE ON A,B.

- A variable trace occurring after a line number trace turns off the line number trace.

A variable trace statement resembles the line number trace statement except that a list of variable names is included. The following example specifies trace output to channel 2, tracing to start at line 340, and tracing of changes in values of A%, element (3,4) of array B, and all of array A$:

```
30      TRACE ON #2%, 340, A%, B(3%, 4%), A$()
```

NOTE

A variable trace of an array cannot be done without first dimensioning the array with a DIM statement.

Variable trace display output takes the following form:

line number identifier type(indices) = new value

Where:

1. Line number is the number of the line in which the variable was assigned a new value.
2. Identifier is the name of the variable.
3. Type is % for integers, $ for strings.
4. Indices identify which element of the array is being traced on and displayed (for array elements only).
5. New value is the new value assigned.

This example shows the result of a trace of an array variable:

TRACE ON A (1, 2) Displays the value of A(1,2) when it is assigned. For example:

220 A(1,2) = 47.3386

This example program illustrates the display resulting from a trace of an integer array program:

```
10      DIM A% (2%, 2%)
20      TRACE ON A%()
30      FOR I% = 0% TO 2%
40      FOR J% = 0% TO 2%
50      A% (I%, J%) = I% * J%
60      NEXT J%
70      NEXT I%
80      TRACE OFF
90      END
```

Results:

```
50  A%(0, 0) = 0
50  A%(0, 1) = 0
50  A%(0, 2) = 0
50  A%(1, 0) = 0
50  A%(1, 1) = 1
50  A%(1, 2) = 2
50  A%(2, 0) = 0
50  A%(2, 1) = 2
50  A%(2, 2) = 4
```

Other Trace Options

TRACE ON line number can be used to define a trace region within a program. The example below traces the array A$ only in the subroutine starting at line 110. Until TRACE OFF is executed, TRACE ON continues to trace all variables for which a TRACE ON was executed, and continues to send trace output to the specified channel or the display.

```
10      DIM A$ (5%, 5%)
20      TRACE ON 110, A$()  ! Start tracing array A$
30                          ! at line 110
40      FOR I% = 0% to 5%
50      A$ (I%, 0%) = CHR$ (ASCII (' ') + I%)
60      GOSUB 110
70      NEXT I%
80      TRACE OFF
90      STOP
100     !
110     FOR J% = 1% TO 5%
120     A$ (I%, J%) = A$(I%, J% - 1%) + CHR$(ASCII('') + I% + J%)
130     NEXT J%
140     TRACE ON 110
150     RETURN
160     END
```

The following trace display output results from running this program. Refer to Appendix G, ASCII/IEEE-1980 Bus Codes, and note the display characters that follow SPACE, character number 32, for clarification of these results.

```
120     A$(0, 1) = !
120     A$(0, 2) = !"
120     A$(0, 3) = !"#
120     A$(0, 4) = !"##
120     A$(0, 5) = !"##%
120     A$(1, 1) = !"
120     A$(1, 2) = !"#
120     A$(1, 3) = !"##
120     A$(1, 4) = !"##%
120     A$(1, 5) = !"##%&
120     A$(2, 1) = "#
120     A$(2, 2) = "##
120     A$(2, 3) = "##%
120     A$(2, 4) = "##%&
120     A$(2, 5) = "##%&'
120     A$(3, 1) = ##
120     A$(3, 2) = ##%
120     A$(3, 3) = ##%&
120     A$(3, 4) = ##%&'
120     A$(3, 5) = ##%&'(
120     A$(4, 1) = $%
120     A$(4, 2) = $%&
120     A$(4, 3) = $%&'
120     A$(4, 4) = $%&'(
120     A$(4, 5) = $%&'()
120     A$(5, 1) = %&
120     A$(5, 2) = %&'
120     A$(5, 3) = %&'(
120     A$(5, 4) = %&'()
120     A$(5, 5) = %&'()*
```

It is also possible to send trace output to different channels. Output is sent to one channel at a time. The following example illustrates this:

```
10      OPEN "TRACE1.DAT" AS NEW FILE 1%! First trace channel
20      OPEN "TRACE2.DAT" AS NEW FILE 2%! Second trace channel
100     TRACE ON A%, B$(), C()          ! Send output to console
250     TRACE ON #1%                    ! Send output to channel 1
370     TRACE ON #2%                    ! Send output to channel 2
1010    TRACE OFF                       ! Discontinue all tracing
1020    END
```

TRACE OFF Statement

Usage: TRACE OFF

TRACE OFF disables any pending or active trace assigned in the program and destroys the variable list.

The following example illustrates that TRACE ON starts only a line number trace after TRACE OFF.

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE OFF                       ! Halt the variable trace
100     TRACE ON                        ! Start a line number trace
```

The following example illustrates a way to suspend tracing until a later point in a program.

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE ON 100                    ! Stop trace until line 100
100     ! Resume tracing variables A and B
```

STOP ON Statement

Usage: STOP ON line number

STOP ON line number, stops execution of a program.

- STOP ON line number, allows a program to be run in sections during logic debugging.
- The program stops at the line number of the STOP when ON line number is not included.
- The program stops at the line number following ON, without executing it, when ON line number is included.
- STOP ON may be executed in either Immediate or Run Mode.
- STOP ON line number enables the STEP command (see below).

STEP Command

Usage: STEP

The Immediate Mode STEP command sets a mode in which each statement within a program is executed individually by pressing RETURN.

- STEP must first be enabled by a breakpoint stop in a running program, caused by STOP ON or CONT TO.
- After a breakpoint stop, type STEP to select Step Mode.
- From Step Mode, type CTRL C or any Immediate Mode command to return to Immediate Mode.
- Any BASIC command or statement that is available in Immediate Mode can also be used to exit Step Mode.
- In Step Mode, one statement is executed each time RETURN is pressed.
- After executing each statement, the display reads: STOP ON LINE n, where n is the next line to be executed.
- When used with a variable TRACE ON, the display will also show changes in selected variables whenever a statement assigns a new variable value.

CONT TO Command

Usage: CONT [TO line number]

The Immediate Mode Continue To line number command causes program execution to continue from a breakpoint stop caused by STOP, STOP ON, CONT TO, or CTRL C.

- The CONT TO line number command is available only in Immediate Mode.
- CONT TO line number must first be enabled by a breakpoint stop in a running program, caused by STOP, STOP ON, CONT TO, or CTRL C.
- Any subsequent action other than entering the CONT TO line number command disables the command. The program must then be rerun to the breakpoint.
- Program execution continues at the statement following the last statement executed.
- When TO line number is included, program execution again stops if the specified line number is encountered, and the statement is not executed.
- CONT TO can be used instead of or in addition to STOP ON and CONT to move quickly through a subroutine or loop that has already been confirmed during Step Mode logic debugging.

The following example program is in main memory during the interaction that follows.

```
10      FOR I% = 1% TO 2%
20      PRINT I% + I%
30      NEXT I%
40      PRINT "Finished."
50      END
```

With the above program in main memory, the following sequence of commands and RETURN entries would produce the responses shown. Programmer entry is shown to the left, and Controller response is shown to the right.

PROGRAMMER ENTRIES	CONTROLLER RESPONSES
	Ready
STOP ON 10	Ready
RUN	Stop at line 10 Ready
STEP	Stop at line 10 Ready
⟨RETURN⟩	Stop at line 20 Ready
⟨RETURN⟩	2 Stop at line 30 Ready
⟨RETURN⟩	Stop at line 20 Ready
⟨RETURN⟩	4 Stop at line 30 Ready
⟨RETURN⟩	Stop at line 40 Ready
⟨RETURN⟩	Done! Stop at line 50 Ready
STOP ON 10	Ready
RUN	Stop at line 10 Ready
CONT TO 40	2 4 Stop at line 40 Ready
CONT	Finished. Ready

Section 16
Error Messages

FLUKE BASIC ERROR LIST

CODE	LEVEL*	EXPLANATION
TYPE: OVERFLOW		
0	F	Memory overflow
1	F	Virtual array file > 64K bytes long, or > 64K elements (XBC)
2	F	Virtual array file too small for arrays
TYPE: SYSTEM		
100	F	BASIC interpreter or Runtime system internal error
101	F	Incompatible lexical file or Extended BASIC program
TYPE: COMMAND		
200	F	Immediate mode error
201	F	Cannot CONTinue
202	F	STEP outside break mode
TYPE: I/O		
300	R	Device not Ready
301	R	Disk write protected
302	R	Illegal channel number specified
303	R	Channel already in use
304	R	Invalid device name or device not present
305	R	File not found on device
306	R	No room on device
307	R	Read/write past end of file
308	R	Channel not open
309	R	RS-232 channel input queue overflow
310	R	Input line too long
311	R	Disk read error
312	R	Illegal filename syntax
313	F	Random access to sequential file
314	F	Sequential access to random file
315	F	Virtual array assigned to sequential device
317	R	Illegal directory on device
318	R	Read (write) from (to) output (input) file
319	R	ON <channel> device not RS-232
320	F	Object file error
321	R	Device directory full
322	R	Illegal operation for device
323	R	File delete protected
324	R	Can't RENAME file
325	R	File medium swapped
326	R	Can't load - too little memory
327	R	Illegal image file format
328	R	Command line too long
329	R	RS-232 port number out of range
330	R	Parallel port number out of range

BASIC ERROR LIST (cont)

CODE	LEVEL*	EXPLANATION
TYPE: INSTRUMENT BUS CONTROL		
400	R	Illegal -488 port number
401	R	Illegal -488 device address
402	R	Illegal -488 secondary device address
403	R	Incomplete -488 handshake
404	R	Too many ports designated for -488 function
405	R	No devices attached to -488 port
406	R	No -488 ports available
407	R	-488 port specified is unavailable
408	R	-488 port timeout
409	R	Illegal WBYTE data
410	R	Parallel poll bit number out of range
411	R	Parallel poll bit sense not 0 or 1
412	R	-488 timeout limit out of range
413	R	TERM string longer than one character
414	R	No -488 driver in System
415	W	SET SRQ status byte value out of range
416	R	Illegal -488 operation for current port state
TYPE: SYNTAX		
500	F	Unrecognized statement
501	F	Illegal character terminating statement
502	F	Illegal subscript (<0)
503	F	Mismatched parentheses
504	F	Illegal let
505	F	Illegal if
506	F	Illegal line number
507	F	Illegal PRINT
508	F	Illegal format for PRINT or NUM$()
509	F	Illegal INPUT statement
510	F	Illegal array dimension size
511	F	Badly formed define
512	F	Illegal FOR statement
513	F	FOR without NEXT
514**	F	NEXT without FOR (jump back into "for" loop)
515	F	Unmatched quotes
516	F	Ill-formed expression
517	F	Bad OPEN statement
518	F	Bad CLOSE statement
519	F	IEEE-488 syntax error
520	F	Initial COM at illegal point in program
521	F	Not a well-structured statement
522	F	Illegal variable name
523	F	ON statement syntax error
524	F	OFF statement syntax error
525	F	TRACE syntax error
526	F	Illegal file size in open

FLUKE BASIC ERROR LIST (cont)

CODE	LEVEL*	EXPLANATION
TYPE: SYNTAX (cont)		
527	F	RENumber parameter error
528	F	RENumber syntax error
529	F	ELSE without IF
530	F	NEXT syntax error
531	F	INPUT WBYTE requires IEEE-488 input
532	F	Illegal subrange descriptor
533	F	WBYTE/RBYTE data not integer type
534	F	Can't specify column for WBYTE/RBYTE subrange
535	F	Can't use undimensioned variable for WBYTE/RBYTE
536	F	Virtual array illegal for WBYTE/RBYTE
537	F	2-dimensional array illegal with WBYTE/RBYTE I/O
538	F	Illegal CONFIG statement
539	F	Illegal RBYTE syntax
540	F	RBYTE increment <=0
541	F	Illegal RBYTE cycle length
542	F	Illegal WBYTE clause syntax
543	F	WBIN/RBIN precision error
544	F	WAIT statement syntax error
545	F	Illegal CALL statement
546	F	Virtual array parameter illegal
547	F	Parameter syntax error
548	F	Illegal SET statement syntax or option
549	F	Require file name for SAVE
550	F	Illegal RENAME statement syntax
TYPE: MATH		
600	F	Illegal mode mixing
601	R	Arithmetic overflow
602	R	Arithmetic underflow
603	R	Divide by zero
604	R	Square root argument > 0
605	R	Exponent too large
606	R	Log argument <=0
607	R	Trig function argument too large
608	R	Illegal argument(s) for power operator
609	F	Illegal floating-point operation code
610	F	Unimplemented floating operation attempted

FLUKE BASIC ERROR LIST (cont)

CODE	LEVEL*	EXPLANATION
TYPE: TRANSFER		
700	F	Illegal GOTO or GOSUB
701	F	RETURN without GOSUB
702	F	RESUME outside interrupt handler
703	F	CALL to undefined FN
704	R	ON {expression} GOTO selector out of range
705	F	CALL to undefined subroutine
706	F	Parameter count mismatch for CALL
707	R	Illegal time/date value
708	R	Timer value not initialized for ON INTERVAL or ON CLOCK
TYPE: INPUT		
800	R	Out of DATA in READ
801	W	Too much data entered for INPUT
802	W	Too little data entered for INPUT
803	W	Illegal character for INPUT or VAL()
804	F	Bad format in data statement
TYPE: VARIABLE		
900	F	Access to undefined variable
901	W	Redimension of array
902	R	Subscript out of range
903	F	COM of variable which is already defined
904	W	String too long for virtual array field
905	F	Incompatible COM declaration
906	F	DIM' within nested interrupt handler
907	F	Bad XOP 1 call
908	F	Illegal array parameter (memory vs. virtual)
909	F	Illegal conformal dimenisoning parameter

* F = Fatal R = Recoverable W = Warning

**A NEXT without FOR error may be caused by exiting a FOR-NEXT loop with a GOTO statement within the loop. If this is executed repeatedly, a fatal user storage overflow error will result. To prevent this from occurring, exit the loop by setting the loop index variable to the maximum value, then use the GOTO statement to branch to the line number containing the NEXT statement.

Here is an example:

```
10 for i% = 0 to 9%
  "
  "
  "
40 if a(I%)  <0 then I% = 9% / goto 100
  "
  "
  "
100 next i%
```

Appendices

CONTENTS

A	Internal Structure of Variables	A-1
B	IEEE-488 Bus Messages	B-1
C	Wbyte Decimal Equivalents	C-1
D	Parallel Poll Enable Codes	D-1
E	Display Controls	E-1
F	Graphics Mode Characters	F-1
G	ASCII & IEEE Bus Codes	G-1
H	Assembly Language Error Handler	H-1
I	Fortran Interface Runtime Library	I-1
J	Virtual Array Dimensioning Program	J-1
K	Graphics	K-1
L	Supplementary Syntax Diagrams	L-1
M	Glossary	M-1
N	Reserved Words	N-1

Appendix A
Internal Structure of Variables

INTRODUCTION

The purpose of this appendix is to provide some information about the means of data storage used by the BASIC processor which may be helpful in applications programming.

VARIABLE STORAGE MEMORY REQUIREMENTS

Each variable used in a BASIC program is entered in one of several symbol tables (depending upon its type) so that, when a reference to the variable occurs, the BASIC interpreter has available the information needed to find the variable's value or address in memory. The amount of space taken by each variable's symbol table entry depends upon the variable type, organization (simple or dimensioned) and where in memory the variable is stored (e.g., COMmon or virtual array). We will call the symbol table information "overhead" in the following paragraphs; the overhead for each type of variable is detailed in Table A-1.

Table A-1. Variable Storage Memory Requirements

Simple variable	4 bytes
Simple variable in COMmon	6 bytes
Dimensioned variable	8 bytes
Dimensioned variable in COMmon	10 bytes
Virtual array	14 bytes

In addition to the symbol table information, additional memory is used to hold the value of the variable itself depending upon the variable type. Table A-2 details the memory requirements for each variable type for all variables EXCEPT those stored in virtual array.

Table A-2. Additional Memory Requirements

Integer	2 bytes
Floating-point	8 bytes
String	4 bytes + 18 bytes for every 16 characters in string

The amount of memory required to represent a variable in main memory (i.e., NOT in a virtual array) may be calculated using the following formula:

m=s+n*l

NOTE

m=total memory required (bytes)

s=symbol table overhead (bytes)

n=number of values represented (1 for a simple variable, or the number of elements in an array).

l=length of a variable of that type (bytes).

The following examples illustrate the process of calculating memory requirements.

1. A simple integer variable requires 6 = 4 + (1*2) bytes.

2. A floating-point array dimensioned to (19,39), which has a total of 800 = (19 + 1)* (39 + 1) elements, would require a total of 6408 = 8 + (800*8) bytes.

3. A simple string (having no characters assigned to it) would require 8 = 4 + (1*4) bytes. If the string were assigned 50 characters, an additional 72 (i.e., 4*18) bytes would be used to hold the value, since for every 16 characters (or for any fragment of the string shorter than 16 characters) a total of 18 bytes is required.

4. A string array dimensioned to (24) contains a total of 25 = 1 + 24 elements. If all elements of the array were set to the null string (having no characters assigned), the memory required would be 108 = 8 + (25*4) bytes. As the elements of the array were assigned non-null values, EACH array element's requirements would have to be calculated using the rule that every 16 characters requires 18 bytes of additional storage.

The memory requirements for a virtual array file may be calculated by the formula:

m=n*v+512

NOTE

m = total memory required (bytes)

n = number of arrays declared in this virtual array file.

v = 14 bytes. This is the symbol table for each array declared within the file.

The requirements for a virtual array file are basically the size of the I/O buffer (512 bytes) which must be created when the file is OPENed (and which disappears when the file is CLOSEd) plus 14 bytes (the symbol table overhead for each array declared within the file) for every array in the file. In the case of multiple DIMension statements referring to the same channel (see section 6 regarding Equivalent Virtual Arrays), only one 512 byte buffer is created, but 14 bytes are still required for every array definition.

I/O BUFFER MEMORY REQUIREMENTS

Every time a OPEN statement is executed, some of the available memory is allocated to provide a temporary data holding area called a buffer. The size of the buffer created is always 512 bytes.

If insufficient memory is available, the BASIC interpreter will report a memory overflow error (error number 0). Whenever a CLOSE statement is executed, the memory used by the buffer is released for other uses.

COMMON MEMORY REQUIREMENTS

The COMmon area is a region of memory holding only the values of variables, i.e., no symbol table data is present. During program chaining, for example, the symbol table is destroyed, but the COMmon area remains intact, so that another program has access to COMmon data via the COM statement. The memory required by the COMmon area may be calculated as:

m=(nr*r)+(ni*i)

NOTE

m=total memory required by COMmon (bytes)

nr=the total number of floating-point values contained in COMmon

r=the length of a floating-point value (8 bytes)

ni=the total number of integer values contained in COMmom

i=the length of an integer value (2 bytes).

The COMmon area will remain intact until the BASIC interpreter ceases execution (via the EXIT or EXEC statement, or when the CTRL/P key is pressed) or until a DELETE ALL statement is executed, which reinitializes the BASIC interpreter.

Appendix B
IEEE-488 Bus Messages

INTRODUCTION

The IEEE-488 standard interface is a bus-structured interconnection method. The bus has sixteen signal lines divided as follows: eight data lines, five bus management lines, and three handshake lines. In addition to these messages lines there are eight ground lines.

DATA LINES

The DATA lines carry raw data, Universal Commands, Address Commands, Addressed Commands, and Device-dependent Commands. Table B-1 presents the various commands which can be sent on the data lines in command mode (determined by the state of the ATN Bus Management Line line). Refer to Appendix G for the actual codes involved.

BUS MANAGEMENT LINES

The bus management lines each have specific management or data transfer control functions. Table B-2 presents the five specific commands which call for some immediate action or flag a condition existing on the interface. Each command corresponds to the bus wire of the same name.

HANDSHAKE LINES

The three handshake lines are used to synchronize data transfers. Table B-3 describes the three handshake lines.

NOTE

Any of these messages may also be given in the "not" form to indicate the reverse message meaning. For instance, DAC would indicate that data has been accepted.

COMMAND MESSAGE SEQUENCES

The various BASIC statements described in the IEEE-488 Bus Input and Output Statements section initiate certain message sequences on the bus. Table B-4 presents these message sequences.

Table B-1. Command Messages

Universal Commands (for all devices)

- LLO-Local Lockout. Disables device LOCAL switches.
- DCL-Device Clear. Clears each device to manufacturer's default status.
- PPU-Parallel Poll Unconfigure. Unconfigures all devices with programmable remote configuration capability.
- SPE-Serial Poll Enable. Causes a device to enter Serial Poll mode.
- SPD-Serial Poll Disable. Disables data I/O lines from Serial Poll status.

Address Commands (or Addresses for each device comparing data I/O lines)

- MLA-My Listen Address. Causes a Device to become a listener.
- UNL-Unlisten. Disables a device from listener status.
- MTA-My Talk Address. Causes a device to become a talker.
- OTA-Other Talk Address. Disables a device from talker status if any other device has received an MTA on data I/O lines 1-5.
- UNT-Untalk. Disables a device from talker status whether or not another talker has been assigned.

Addressed Commands (For Addressed-to-Listen devices)

- GTL-Go To Local. Returns device from REMOTE mode.
- SDC-Selected Device Clear. Clears a device to manufacturer's default status.
- PPC-Parallel Poll Configure. Sets up device to be configured with the PPE or PPD messages.
- GET-Group Execute Trigger. Initiates a device function to a selected, addressed group of devices.
- TCT-Take Control. A talker device becomes a Controller In Charge.

Table B-1. Command Messages (cont)

Secondary Commands (for all devices)

- MSA-My Secondary Address. Follows an MTA or MLA to allow the device extended listener or talker capability.
- OSA-Other Secondary Address. Like an MSA except will not unaddress if the first MTA equalled its address switch settings.
- PPD-Parallel Poll Disable. Unconfigures device.
- PPE-Parallel Poll Enable Configures device.

Device-Dependent Commands

- DAB-Data Byte. Any byte on the data I/O lines transferred between a talker and one or more listeners.
- EOS-End of String. Line feed or carriage return written across the data I/O lines to indicate an end of a logical string.
- NUL-Null. Zeros across the data I/O lines to initialize the bus.
- PPR-Parallel Poll Response. Response to a request for a Parallel Poll.
- STB-Status Byte. Response to an SPE (Serial Poll Enable). Concurrent with RQS.
- SRQ-Service Request. Response to an SPE if Service is requested.

Table B-2. Bus Management Lines

- ATN-Attention. Specifies how data is to be interpreted.
- IFC-Interface Clear. Resets interface of all devices to a known state.
- REN-Remote Enable. Prepares all devices for accepting remote (from the Controller) commands.
- EOI-End or Identify. Indicates that a talker device has ended a multi-byte transfer or that a controller is in a polling sequence.
- SRQ-Service Request. Indicates that a device wants to interrupt the current sequence of events for attention.

Table B-3. Handshake Lines

- DAV-Data Valid. Indicates that a device has available and valid data on the DATA line.
- NRFD-Not Ready for Data. Indicates that a listener device is not ready to accept data.
- NDAC-Not Data Accepted. Indicates that a listener device has not yet accepted data.

Table B-4. BASIC IEEE-488 Bus Command Messages Sequences

BASIC COMMAND	MESSAGE SEQUENCE
CLEAR (by device)	$\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device) SDC
CLEAR (by port)	$\overline{\text{EOI}}$ ATN UNL UNT DCL
CONFIG (by device for unconfigure)	$\overline{\text{EOI}}$ ATN UNL UNT MLA PPD UNL
CONFIG (to line with sense)	$\overline{\text{EOI}}$ ATN UNL UNT MLA PPC PPE UNL

Table B-4. BASIC IEEE-488 Bus Command Messages Sequences (cont)

BASIC COMMAND	MESSAGE SEQUENCE
INIT (by port)	REN ATN IFC IFC (after 100 usec wait) EOI ATN UNL UNT PPU
INPUT LINE WBYTE	(output WBYTE data) EOI
NOTE *Issued only if device specified with INPUT statements.*	
	ATN UNL UNT MTA ATN (accept data byte) (output WBYTE data) EOI ATN (accept data byte)
	(output WYBTE data) EOI ATN (accept data)
LOCAL (by device)	EOI ATN UNL UNT MLA (for each device) GTL
LOCAL (by port)	REN

Table B-4. BASIC IEEE-488 Bus Command Messages Sequences (cont)

BASIC COMMAND	MESSAGE SEQUENCE
LOCKOUT (by port)	REN EOI ATN LLO
PASSCONTROL	ATN UNL UNT MTA TCT ATN
PPL	ATN EOI (Accept poll status) EOI
PRINT USING	EOI ATN UNL
NOTE *Issued only if device(s) specified in PRINT statement.*	
	MLA (for each device) ATN (output data)
RBYTE	EOI ATN (accept data)
REMOTE (by device)	REN EOI ATN UNL UNT MLA (for each device
REMOTE (by port)	REN

Table B-4. BASIC IEEE-488 Bus Command Messages Sequences (cont)

BASIC COMMAND	MESSAGE SEQUENCE
SPL	$\overline{\text{EOI}}$ ATN UNL UNT MTA SPE $\overline{\text{ATN}}$ (accept status byte) ATN UNT SPD
TRIG	$\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device) GET
WBYTE	EOI (set as required) ATN (set as required) (output data byte) EOI (set as required) ATN (set as required) (output data byte) . . .

Appendix C
WBYTE Decimal Equivalents

NOTE

Refer to the WBYTE discussion in the IEEE-488 Bus Input and Output Statements. Use these decimal values when building up an array of data bytes to be output to the IEEE-488 Bus.

Table C-1. Decimal Equivalents

BUS MESSAGE	DECIMAL EQUIVALENT
EOI	256
ATN	512
GTL	513
GET	520
DCL	532
UNL	575
UNT	607

Table C-2. Address Messages

DEVICE ADDRESS	MLA	MTA	MSA	DEVICE ADDRESS	MLA	MTA	MSA
0	544	576	608	16	560	592	624
1	545	577	609	17	561	593	625
2	546	578	610	18	562	594	626
3	547	579	611	19	563	595	627
4	548	580	612	20	564	596	628
5	549	581	613	21	565	597	629
6	550	582	614	22	566	598	630
7	551	583	615	23	567	599	631
8	552	584	616	24	568	600	632
9	553	585	617	25	569	601	633
10	554	586	618	26	570	602	634
11	555	587	619	27	571	603	635
12	556	588	620	28	572	604	636
13	557	589	621	29	573	605	637
14	558	590	622	30	574	606	638
15	559	591	623	31	575	607	639

Appendix D
Parallel Poll Enable Codes

NOTE

Refer to the CONFIG statements discussion before using Table D-1.

Table D-1. Parallel Poll Enable Codes

	RESPONSE LINE	DECIMAL
SENSE "0"	DI01	96
	DI02	97
	DI03	98
	DI04	99
	DI05	100
	DI06	101
	DI07	102
	DI08	103
SENSE "1"	DI01	104
	DI02	105
	DI03	106
	DI04	107
	DI05	108
	DI06	109
	DI07	110
	DI08	111

Appendix E
Display Controls

DISPLAY CONTROLS

The two tables in this section list the display control sequences for the display contained in the Instrument Controller. Table E-1 describes the control sequences used to control the visual attributes of the display. Table E-2 describes the ANSI Standard Selective Parameters used by the Instrument Controller.

Table E-1. Display Control Sequences

FUNCTION	CONTROL SEQUENCE	COMMENTS
Cursor Controls		
Up n lines	⟨ESC⟩[nA	The cursor stops at the edge if the number given as an argument results in movement past the edge of the screen.
Down n lines	⟨ESC⟩[nB	
Right n columns	⟨ESC⟩[nC	
Left n columns	⟨ESC⟩[nD	
Direct to line, column	⟨ESC⟩[l; c H	
Scroll down one line	⟨ESC⟩D	
Scroll up one line	⟨ESC⟩M	
Scroll to start new line	⟨ESC⟩E	
Erasing		
To end of display	⟨ESC⟩[J or ⟨ESC⟩[0J	
To start of display	⟨ESC⟩[1J	
All of display	⟨ESC⟩[2J	
To end of line	⟨ESC⟩[K or ⟨ESC⟩[0K	
To start of line	⟨ESC⟩[1K	
All of line	⟨ESC⟩[2K	
Attributes		
Attributes Off*	⟨ESC⟩[m or ⟨ESC⟩[0m	
High Intensity	⟨ESC [1m	
Underline	⟨ESC [4m	
Blinking	⟨ESC⟩[5m	
Reverse Image	⟨ESC⟩[7m	
Non-destructive character	⟨ESC⟩=	Replaces any attributes in effect at the position that it is directed to (via the CPOS function, for example).

Table E-1. Display Control Sequences (cont)

FUNCTION	CONTROL SEQUENCE	COMMENTS
Cursor Status		
Request cursor position	⟨ESC⟩[6n	For a program to make use of the report, a logical input channel must exist between the program and KB0:
Cursor position report	⟨ESC⟩[l, c R	
Size of Characters		
Normal*	⟨ESC⟩[p or ESC [0p	These commands affect the entire display.
Double	⟨ESC⟩1p	
Character Graphics		
Disabled*	⟨ESC⟩[2p	These commands also affect the graphics plane.
Enabled	⟨ESC⟩[3p	
Keyboard		
Enabled*	⟨ESC⟩[4p	Even when disabled, the keyboard can respond to control codes. To exit a locked condition, use ⟨CTRL⟩/T to unlock the keyboard, reset the screen to normal-size characters, home the cursor (upper left), and disable the graphics plane.
Disabled	⟨ESC⟩[5p	
Cursor Type		
Blinking Underscore*	⟨ESC⟩[0x	
Steady Underscore	⟨ESC⟩[1x	
Blinking Block	⟨ESC⟩[2x	
Steady Block	⟨ESC⟩[3x	

*Indicates the default conditions.

Table E-2. Selective Parameter Display Controls

FUNCTION	MODE SELECTION	EQUIVALENT NUMERIC SEQUENCE
Attribute Mode		No equivalent.
Field*	⟨ESC⟩[?1l	
Character	⟨ESC⟩[?h	
Character Size		
Normal*	⟨ESC⟩[?2l	⟨ESC⟩[0p
Double size	⟨ESC⟩[?2h	⟨ESC⟩[1p
Character Graphics		Similar to ⟨ESC⟩[2p and ⟨ESC⟩[3p, except that these commands do not affect the graphics plane.
Disable*	⟨ESC⟩[3l	
Enable	⟨ESC⟩[3h	
Keyboard		
Unlocked*	⟨ESC⟩[?4l	⟨ESC⟩[4p
Locked	⟨ESC⟩[4h	⟨ESC⟩[5p
Opaque to Graphics*	⟨ESC⟩[?5l	When this command is received, any graphics displays that cross display character cells are hidden behind the character cell, an area 8 pixels wide and 14 high. This mode is used to make characters stand out from surrounding graphics displays. There is no equivalent capability for the 1720A Controller.
Transparent to Graphics	⟨ESC⟩[?5h	This mode causes displayed characters to be transparent to the graphics display. Any graphics displays that cross a character cell are not obstructed by the cell. Select this mode to blend characters into the graphics display.

Table E-2. Selective Parameter Display Controls

FUNCTION	MODE SELECTION	EQUIVALENT NUMERIC SEQUENCE
Character Display		No equivalents.
Disable	⟨ESC⟩[?6l	
Enable*	⟨ESC⟩[?6h	
Graphics Plane		No equivalents.
Disable	⟨ESC⟩[?7l	
Enable*	⟨ESC⟩[?7h	
Cursor Display		No equivalents.
Disable	⟨ESC⟩[?8l	
Enable*	⟨ESC⟩[?8h	

*Indicates the default conditions.

NOTES: 1. Save typing by predefining ESC [as the following string:

```
ES$=CHR$(27)+"["
```

2. Multiple enhancement or mode commands are separated by semicolons.

```
Example: PRINT ES$;"1;5;7p";
```

3. Display controls are introduced by "ESCape[" (scrolling controls do not use "[".) Any method that sends the required character sequence to the display will give the above results. For example, if a data file is created of "ESC[" character sequences, typing the name of the file in FUP mode will cause the display response.

Appendix F
Graphics Mode Characters

1722A DISPLAY RESPONSE

CHARACTER	NORMAL SIZE	DOUBLE SIZE	FUNCTION
0			Top Right Corner
1			Top Left Corner
2			Bottom Right Corner
3			Bottom Left Corner
4			Top Intersect
5			Right Intersect
6			Left Intersect
7			Bottom Intersect
8			Horizontal Line
9			Vertical Line
:			Crossed Line

NOTES:

1. To enable Graphics Mode, send the display ESC[3p or ESC [?3h
2. To disable Graphics Mode, send the display ESC[2p or ESC [?3l
3. In Graphics Mode, characters in the left column are displayed as shown.
4. Use the character names as defined to construct illustrations that do not change form between normal and double size.

Appendix G
ASCII/IEEE-488 Bus Codes

FLUKE®

ASCII & IEEE-488 BUS CODES

B7 B6 B5 B4 / BITS B3 B2 B1 B0	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
	CONTROL		NUMBERS SYMBOLS		UPPER CASE		LOWER CASE	
0 0 0 0	0 0 NUL NUL	16 10 ρ DLE	32 20 SP SPACE MLA0	48 30 0 0 MLA16	64 40 @ @ MTA0	80 50 P P MTA16	96 60 ` ` MSA0	112 70 p P MSA16
0 0 0 1	1 1 β SOH GTL	17 11 σ DC1	33 21 ! ! MLA 1	49 31 1 1 MLA17	65 41 A A MTA1	81 51 Q Q MTA17	97 61 a a MSA1	113 71 q q MSA17
0 0 1 0	2 2 γ STX	18 12 τ DC2	34 22 " " MLA2	50 32 2 2 MLA18	66 42 B B MTA2	82 52 R R MTA18	98 62 b b MSA2	114 72 r r MSA18
0 0 1 1	3 3 δ ETX	19 13 υ DC3	35 23 # # MLA3	51 33 3 3 MLA19	67 43 C C MTA3	83 53 S S MTA19	99 63 c c MSA3	115 73 s s MSA19
0 1 0 0	4 4 ε EOT SDC	20 14 φ DC4 DCL	36 24 $ $ MLA4	52 34 4 4 MLA20	68 44 D D MTA4	84 54 T T MTA20	100 64 d d MSA4	116 74 t t MSA20
	5 5	21 15	37 25	53 35	69 45	85 55	101 65	117 75

1	0	0	0	8 8 **BS** BS GET	24 18 **Ω** CAN SPE	40 28 **(** (MLA8	56 38 **8** 8 MLA24	72 48 **H** H MTA8	88 58 **X** X MTA24	104 68 **h** h MSA8	120 78 **x** x MSA24
1	0	0	1	9 9 **HT** HT TCT	25 19 **√** EM SPD	41 29 **)**) MLA9	57 39 **9** 9 MLA25	73 49 **I** I MTA9	89 59 **Y** Y MTA25	105 69 **i** i MSA9	121 79 **y** x MSA25
1	0	1	0	10 A **LF** LF	26 1A **→** SUB	42 2A ***** * MLA10	58 3A **:** : MLA26	74 4A **J** J MTA10	90 5A **Z** Z MTA26	106 6A **j** j MSA10	122 7A **z** z MSA26
1	0	1	1	11 B **VT** VT	27 1B **ESC** ESC	43 2B **+** + MLA11	59 3B **;** ; MLA27	75 4B **K** K MTA11	91 5B **[** [MTA27	107 6B **k** k MSA11	123 7B **{** { MSA27
1	1	0	0	12 C **FF** FF	28 1C **↑** FS	44 2C **,** , MLA12	60 3C **<** < MLA28	76 4C **L** L MTA12	92 5C **** \ MTA28	108 6C **l** l MSA12	124 7C **¦** ¦ MSA28
1	1	0	1	13 D **CR** CR	29 1D **÷** GS	45 2D **-** - MLA13	61 3D **=** = MLA29	77 4D **M** M MTA13	93 5D **]**] MTA29	109 6D **m** m MSA13	125 7D **}** } MSA29
1	1	1	0	14 E **S0** S0	30 2E **Σ** RS	46 2E **.** . MLA14	62 3E **>** > MLA30	78 4E **N** N MTA14	94 5E **∧** ∧ MTA30	110 6E **n** n MSA14	126 7E **~** ~ MSA30
1	1	1	1	15 F **SI** SI	31 1F **≈** US	47 2F **/** / MLA15	63 3F **?** ? UNL	79 4F **O** O MTA15	95 5F **—** — UNT	111 6F **o** o MSA15	127 7F ■ RUB-OUT MSA31
2^3	2^2	2^1	2^0	ADDRESSED COMMANDS	UNIVERSAL COMMANDS	LISTEN ADDRESSES		TALK ADDRESSES		SECONDARY ADDRESSES OR COMMANDS	

KEY

decimal

38	26
&	

hex

1722A DISPLAY

P/N 016020-NOVEMBER 1986

Appendix H
Assembly Language Error Handler

INTRODUCTION

The assembly error handler provides a means of reporting error conditions to the BASIC interpreter program when using an Assembly language subroutine with BASIC.

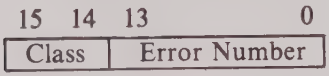
USING A USER-SPECIFIED ERROR HANDLER SUBROUTINE

A user-specified error handler subroutine can use a form of XOP 0 to get the BASIC error handler to report and to attempt recovery from errors. This causes the BASIC error handler to treat the error as appropriate to an error of the given recovery class (for example, Fatal, Recoverable, or Warning Only).

The instruction format is as follows:

XOP @⟨error number & recovery class⟩,0

Where error number & recovery class is encoded as follows:



1. Class is encoded as:

   0: Fatal
   1: Recoverable
   2: Warning only

2. Error Number can be any error number desired.

   a. Error numbers from 0 to 999 are already assumed to have predefined meanings.

   b. Error numbers greater than 999 have the code "User" printed as the error code. An example of a "User" code instruction and the resulting message are as follows:

```
XOP     @1000,0
!User error 1000 at line (line number of the CALL statement)
```

EXAMPLE

The following example illustrates one approach to detecting error conditions within an Assembly language subroutine. The example program computes the RSS (root-sum-squares) of two passed parameters (A & B), returning the answer in C and an error flag in R%.

Here is the example program:

```
                    def     DET
*
        ref         F$RGMY
*
DET     data        ws,det              ;transfer vector
        idt         'det'
*
det     equ         $                   ;entry point
        bl          @F$RGMY             ;get the parameters
        data        4                   ;
        data        r4*2+ws,            ;put them in r4 through r7
        text        'DET                ;name
        clr         r8                  ;clear overflow flag
        ld          *r4                 ;get x
        md          r0                  ;x*x
.       if          ov                  ;overflow
        seto        r8                  ;set flag
.       endif
        std         @temp               ;save
        ld          *r5                 ;get y
        md          r0                  ;get y*y
.       if          ov                  ;overflow
        seto        r8                  ;set flag
.       endif
        ad          @temp               ;x*x + y*y
.       if          ov                  ;overflow
        seto        r8                  ;set flag
.       endif
        dsqr                            ;SQRT(x*x + y*y)
        std         *r6                 ;return result
        mov         r8,*r7              ;return error flag
        rtwp                            ;done
*
        ws          bss     32          ;workspace
        bss         4
        data        1                   ;reentrant
        temp        bss     8           ;temporary
        end
*
```

This is the subroutine call from BASIC:

```
 10       A = 4 \ B = 3 \ C = 0\ R% = 0%
 20       CALL DET(A, B, C, R%)
 30       IF R% <> 0% GOTO 100     !goto error handler
 40       .
 50       .
 60       .
100       ! this is the error handler
```

Appendix I
FORTRAN Interface Runtime Library

FORTRAN INTERFACE RUNTIME LIBRARY

The FORTRAN Interface Runtime Library provides a link between the BASIC Interpreter and FORTRAN subroutines used with BASIC. The Fortran Interface Runtime Library implements most of the FORTRAN language elements and also includes explicit parameter-passing subroutines for string variable exchange and error message exchange. The language elements available for FORTRAN subroutines are described in this appendix.

The following restrictions apply:

- No I/O capability (such as a FORTRAN READ statement) is provided. Attempts to use I/O will result in unresolved global errors during the linking process.

- Only the data types which are compatible with BASIC are allowed:

  Double-Precision Floating Point

  Integer

  Logical

Attempting to use other data types will result in unresolved global errors during the linking process.

- Only the set of statements, intrinsic functions, and external functions listed below under Available FORTRAN Language Elements are used.

AVAILABLE FORTRAN LANGUAGE ELEMENTS

FORTRAN programs normally run under control of the FORTRAN Runtime System program, which forms a link between the FORTRAN program and the operating system program of the Instrument Controller. Interpreted BASIC programs run under the control of its own runtime system programs. In order for an interpreted BASIC program to run subroutines which are written in FORTRAN, the Fortran Interface Runtime Library of subroutines is provided which implements the FORTRAN language elements normally provided by the FORTRAN Runtime System program. Whenever a FORTRAN subroutine is used, this library must be linked with it.

The FORTRAN Interface Runtime Library does not include all of the FORTRAN language elements. The statements, external functions, and internal functions that are available are listed in tables below. Refer to the FORTRAN Programmer's Reference Manual for a complete description of the FORTRAN language.

The following FORTRAN statements are allowed in subroutines that will be used by BASIC (with the support of the FORTRAN Interface Runtime Library):

ASSIGN	DO	INTEGER
BLOCK DATA	DOUBLE PRECISION	INTEGER*2
CALL	END	IMPLICIT
COMMON*	EQUIVALENCE	LOGICAL
CONTINUE	EXTERNAL**	REAL*8
COPY	FUNCTION**	RETURN
DATA	GO TO	SUBROUTINE
DIMENSION	IF	IF

*The COMMON area in a FORTRAN program is separate from the COM area in BASIC programs.

**Note that, although EXTERNAL and FUNCTION are allowed, they only work between FORTRAN subroutines, not between FORTRAN subroutines and BASIC programs.

The following external functions are allowed in FORTRAN subroutines that will be used by BASIC programs:

DATAN	DSIN	IOR
DCOS	DSQRT	ISHFT
DEXP	IBCLR	LAND
DLOG	IBSET	NOT
DLOG10	IBTEST	
DMOD	IEOR	

The following intrinsic functions are allowed in FORTRAN subroutines that will be used by BASIC programs:

DABS	DSIGN	MAX0
DMAX0	IABS	MIN0
DMAX1	IDIM	MOD
DMIN0	IDINT	
DMIN1	ISIGN	

All standard FORTRAN string manipulation functions are available. These subroutines are as follows:

DNCASE	INDEX	IVERFY
KOMSTR	LENGTH	MFLD
SUBSTR	TRANS	UPCASE

USER SUBROUTINES

Three FORTRAN subroutines are provided to allow the programmer to exchange string and error information between BASIC and FORTRAN subroutines. The subroutines are:

GETSTR - Get a string from a BASIC program.

PUTSTR - Send a string to a BASIC program.

ERROR - Send an error sequence to a BASIC program.

Two of these subroutines are used for passing strings. The GETSTR subroutine passes a string from a BASIC program into a linear array in a FORTRAN routine. The PUTSTR subroutine passes a string from a linear array in a FORTRAN routine Back to a string in a BASIC program.

The ERROR subroutine provides the capability of passing error information from a FORTRAN subroutine back to a BASIC program. An error number and a severity code is sent which the BASIC program uses to process the error. The error number allows programmers to define a private error list. Severity codes are sent to define the appropriate program response for Fatal, Recoverable, and Warning level errors.

The following example shows how to use the string transfer subroutines in this library to pass string variables to and from a BASIC program. Refer to the descriptions of the GETSTR and PUTSTR subroutines in the FORTRAN Interface Runtime Library Users Manual and the DNCASE subroutine in Appendix D of the FORTRAN Programmers Reference Manual, and refer to the description of the CALL statement in this manual while studying this example.

The calling statement in the BASIC program is:

```
100       !convert from uppercase to lowercase
110       !input in A$, output in B$, error code in ER%
120       CALL UPLOW(A$,B$,ER%)
```

and the FORTRAN subroutine where the library subroutines are used is:

```
      SUBROUTINE UPLOW(A,B,ERR)
      INTEGER A(1),B(1),ERR,LEN,STRING(40)
      LEN=80
      CALL GETSTR(A,STRING,LEN)
      IF (LEN .EQ. -1) GOTO 9000
      CALL DNCASE(STRING,LEN,IERR)
      IF (IERR .NE. 0) GOTO 9000
      CALL PUTSTR(STRING,B,LEN)
      ERR=0
      RETURN

9000  ERR=-1
      RETURN
```

The CALL statement in the BASIC program branches to a FORTRAN subroutine named UPLOW and passes the location of three variables, two strings, A$ and B$, and an integer ER%.

The first line of the FORTRAN subroutine defines the subroutine name UPLOW and the names of the variables which are exchanged with the BASIC program. Since these variables are passed by location reference and not by name, the FORTRAN subroutine uses these three variables with its own names. A$ in the BASIC program will be called A in the FORTRAN subroutine; B$ will be called B; and ER% will be called ERR. The next program line declares the variable types: single-dimensioned integer arrays for the two-byte string names, two integers for an error log and a length counter, and an 80-byte integer array STRING to be used for the string in the FORTRAN subroutine. In the third line, the LEN integer is set to 80, the length of the target array.

The FORTRAN subroutine GETSTR is called to transfer a string from the BASIC program to the FORTRAN subroutine. In this case, the BASIC string A$ (FORTRAN variable A) is transferred to the FORTRAN variable STRING. Following this, the LEN variable is examined to determine if it contains a "target too large" error code (–1), in which case control is transferred to line 9000 and subsequently back to the BASIC program with ERR=–1.

If no error code is detected, the standard FORTRAN character manipulation subroutine DNCASE is used to convert the uppercase string to a lowercase string. Note that LEN contains the string length in bytes if the GETSTR procedure was successful.

After once again checking for an error code (IERR<>0), the PUTSTR subroutine is used to return STRING to the the BASIC program (as B$ or FORTRAN B). The error code variable ERR (ER% in the BASIC) is set to 0 to indicate a successful procedure, and control is returned to the BASIC program.

After control returns to the the BASIC program, the variable B$ contains the lowercase version of the variable A$, and the variable ER% contains a 0.

SUBROUTINE FORMAT

The following example illustrates the use of the CALL statement of the BASIC to use a FORTRAN subroutine. Additional information can be found in the Fortran Interface Runtime Library User Manual.

the BASIC Program:

```
10   DIM A(999%)                                  !DIMENSION ARRAY
20   LINK 'STAT'                                  !FORTRAN SUBROUTINE
30   FOR I%=0% TO 999%\A(I%)=RND\NEXT I%          !LOOP, FILL ARRAY
40   M = 0 \ S = 0                                !RESET VARIABLES
50   CALL STAT(A(),100%,M,S)                      !CALL SUBR.,PASS VARIABLES
60   PRINT 'The mean value is ';M                 !PRINT MEAN
70   PRINT 'The standard deviation is ';S! PRINT STD. DEV.
80   END
```

FORTRAN Subroutine

```
C
C     This is an example of a FORTRAN subroutine called from a
C     the BASIC program.
C
C     This subroutine calculates the mean and standard deviation
C     of an array of double-precision values.
C

      SUBROUTINE STAT(ARRAY, N, MEAN, STDEV)
C
      REAL *8 ARRAY(1), MEAN, STDEV, SUM, SUMSQ
      INTEGER I,N
C
      SUM = 0
      SUMSQ = 0
      DO 10 I=1,N
              SUM = SUM + ARRAY(I)
              SUMSQ = SUMSQ + ARRAY(I)*ARRAY(I)
10     CONTINUE
      MEAN = SUM/N
      STDEV = DSQRT(SUMSQ/N - MEAN*MEAN)
      RETURN
      END
```

The following command file will generate a FORTRAN module which can be linked with the BASIC. The file runs the FORTRAN Compiler program, then the Linkage Editor program, then the Object Translator program. Finally, the File Utility Program is run, the object file deleted, and the memory packed. Control is then returned to the Console Monitor.

```
COMMAND

FC
EDO:?,KBO:,EDO:=?

LE ?,EDO:?,EDO:

TASK ?
FORM ASCII
PARTIAL
INCL EDO:?
FIND FTN$IF
END

OTP

?=?

FUP
EDO:?,OBJ/D
EDO:/P
/X
```

```
COMMENT

RUN FORTRAN COMPILER PROGRAM
PLACE THE OBJECT AND SOURCE FILES
ON EDO:  AND THE LIST ON KBO:
RUN THE LINKAGE EDITOR,  PLACE THE
OUTPUT AND SCRATCH FILES ON EDO:
CREATE PHASE 0
CREATE ASCII FORMAT OUTPUT FILE
SPECIFY PARTIAL LINKAGE
LOAD THE INPUT FILE
LOAD THE FORTRAN INTERFACE MODULE
EXECUTE COMMANDS AND EXIT THE
LINKAGE EDITOR PROGRAM
RUN THE OBJECT TRANSLATOR UTILITY
PROGRAM
TRANSLATE THE FILE TO BASIC
COMPATIBLE FORM.
RUN THE FILE UTILITY PROGRAM
DELETE THE OBJECT FILE
PACK THE E-DISK
EXIT BACK TO CONSOLE MONITOR
PROGRAM
```

Appendix J
Virtual Array Dimensioning Program

```
10  ! EXAMPLE 14
20  ! VERSION 1.0, 30 DECEMBER 1981
25  !
26  ! THIS PROGRAM CALCULATES THE SEQUENCE OF VIRTUAL ARRAY DIMENSIONING
27  ! WHICH WILL REQUIRE THE LEAST AMOUNT OF DISK STORAGE
28  !
30  PRINT CHR$(27%)+"[2J"  ! CLEAR DISPLAY
40  !
50  L$="SHHH,HH"           ! PRINT USING FORMAT FOR BLOCKS
60  Y$="SHHHH"             ! PRINT USING FORMAT FOR BYTES
70  !
100 ! DEFINITION OF VARIABLES
110 !
120 ! (Y) STANDS FOR B(Y)TE IN THE VARIABLE NAMES
130 ! (L) STANDS FOR B(L)OCK IN THE VARIABLE NAMES
140 !
150 ! A% = ORDER SUBSCRIPT
160 ! B% = ORDER SUBSCRIPT
170 ! C% = ORDER SUBSCRIPT
180 ! K% = COUNTER FOR THE SIX POSSIBLE SEQUENCES
190 ! I% = ARRAY NO.
200 ! J% = ELEMENT COUNTER
210 ! L% = BLOCK COUNTER
220 ! N% = TOTAL NO. OF ARRAYS TO BE DIMENSIONED
230 ! Y% = BYTE COUNTER
240 !
250 ! AY% = ACTUAL TOTAL BYTES USED BY ALL THE ARRAYS
260 ! TY% = MINIMUM TOTAL BYTES REQUIRED FOR ALL THE ARRAYS
270 ! AL  = ACTUAL TOTAL BLOCKS USED BY ALL THE ARRAYS
280 ! TL  = MINIMUM TOTAL BLOCKS REQUIRED FOR ALL THE ARRAYS
290 ! MNU = MINIMUM UNUSED BYTES
300 !
310 ! EL%(I%) = ENDING BLOCK NO. FOR ARRAY I%
320 ! EY%(I%) = ENDING BYTE NO. WITHIN BLOCK NO. EL%(I%) FOR ARRAY I%
330 ! NE%(I%) = NO. OF ELEMENTS IN ARRAY I%
340 ! NY%(I%) = NO. OF BYTES PER ELEMENT IN ARRAY I%
350 ! SL%(I%) = STARTING BLOCK NO. FOR ARRAY I%
360 ! SQ$(K%) = SEQUENCE OF DIMENSIONING
370 ! SY%(I%) = STARTING BYTE NO. WITHIN BLOCK NO. SL%(I%) FOR ARRAY I%
380 ! SZ%(I%) = TOTAL NO. OF BYTES REQUIRED BY ARRAY I%
390 ! UB%(K%) = UNUSED BYTES FOR THE KTH SEQUENCE TEST
400 ! UY%(I%) = ACCUMULATOR OF NO. OF UNUSED BYTES IN BLOCKS USED BY ARRAY I%
410 ! AP$(1..N%) = ARRAY POSITION
415 !
420 !***************** PICK OUTPUT DEVICE ***************
430 !
440 CLOSE 1
450 OPEN "KBO:" AS NEW FILE 1
460 !
500 !***************** INITIALIZATION ***************
510 !
520 N%=3%       ! NO. OF ARRAYS TO BE DIMENSIONED
525 NF%=3%*2*1% ! = N% FACTORIAL POSSIBLE DIMENSIONING COMBINATIONS
530 RL%=8%      ! NO. OF BYTES REQUIRED BY A REAL NUMBER
540 NT%=2%      ! NO. OF BYTES REQUIRED BY AN INTEGER
550 N1%=2%      ! STRING LENGTH OF 2 CHARACTERS
570 N2%=4%      !                  4
575 N3%=8%      !                  8
580 N4%=16%     !                 16
590 N5%=32%     !                 32
600 N6%=64%     !                 64
610 N7%=128%    !                128
620 N8%=256%    !                256
630 N9%=512%    !                512
640 !
650 DIM SY%(N%),SL%(N%),EY%(N%),EL%(N%),UY%(N%),NE%(N%),NY%(N%),SZ%(N%)
660 DIM SQ$(NF%), UB%(NF%), AP$(N%)
670 K%=0
680 !
700 !********** ASSIGN SUBSCRIPTS FOR THE SIX POSSIBLE SEQUENCES *******
710 !
720 T1$="          SUBSCRIPTS WHICH IDENTIFY ARRAYS"
730 T2$="          ---------------------------------"
740 T3$="ARRAY NO.   1         2         3          (ORDER:  123)"
750 T4$=" DIM #1, A%(1%,4%), E(10%,9%), Q$(10%)=64%   ! ARRAYS BEING TESTED"
760 PRINT #1, T1$ \ PRINT #1, T2$ \ PRINT #1, T3$ \ PRINT #1, T4$ \ PRINT #1
770 !
```

```
780 FOR A%=1% TO N%              ! A% = POSITION OF ARRAY NO. 1
790 FOR B%=1% TO N%              ! B% = POSITION OF ARRAY NO. 2
800   IF B%=A% GOTO 2180
810 FOR C%=1% TO N%              ! C% = POSITION OF ARRAY NO. 3
820   IF C%=A% OR C%=B% GOTO 2170
825     AP$(A%)="1" \ AP$(B%)="2" \ AP$(C%)="3"   ! ARRAY POSITION STRINGS
830     K%=K%+1%                                 ! INCREMENT TEST COUNTER (1% TO NF%)
840     SQ$(K%)=AP$(1%)+AP$(2%)+AP$(3%)          ! STORE ACTUAL SEQUENCE
850 !
860 ! SUBSCRIPTS IN THE SIX VARIABLES BELOW CORRESPOND
870 ! WITH THE SELECTED DIMENSIONED ORDER BEING TESTED
880 !
890 ! NO. OF ELEMENTS -- BYTES PER ELEMENT -- ORDER --          --  ARRAY NO.
900 !
910     NE%(A%)=10%     \     NY%(A%)=N1%    ! A%TH ARRAY TO BE DIMENSIONED (1)
920     NE%(B%)=110%    \     NY%(B%)=RL%    ! B%TH ARRAY TO BE DIMENSIONED (2)
930     NE%(C%)=11%     \     NY%(C%)=N6%    ! C%TH ARRAY TO BE DIMENSIONED (3)
940 !
950 ! NOTE THERE ARE THREE FACTORIAL POSSIBLE DIMENSION COMBINATIONS
960 ! FOR THE ABOVE THREE ARRAYS: 123, 132, 213, 231, 312, 321
970 !
1000 !**************** MAIN PROGRAM ************************
1010 !
1020 Y%=0          ! INITIALIZE BYTE COUNTER
1030 L%=1%         ! INITIALIZE BLOCK COUNTER
1040 UB%(K%)=0     ! INITIALIZE TOTAL UNUSED BYTE ACCUMULATOR
1050 !
1060 !                    ---- DISPLAY HEADINGS -----
1070 !
1080 PRINT #1,TAB(34);"ORDER: ";SQ$(K%)
1090 PRINT #1,TAB(9);"ARRAY";TAB(16);"STARTING";TAB(29);"STARTING";
1100 PRINT #1,TAB(44);"ENDING";TAB(57);"ENDING";TAB(70);"UNUSED"
1110 PRINT #1,TAB(9);"NO.";TAB(18);"BYTE";TAB(30);"BLOCK";
1120 PRINT #1,TAB(45);"BYTE";TAB(57);"BLOCK";TAB(70);"BYTES"
1130 !
1140 !                ---- ALLOCATE STORAGE TO BLOCKS ----
1150 !
1160 FOR I%=1% TO N%                  ! ARRAY NO. COUNTER
1170   SZ%(I%)=NE%(I%)*NY%(I%)        ! TOTAL BYTES REQUIRED BY THE ARRAY
1180   SY%(I%)=Y%+1%                  ! STARTING BYTE NO.
1190   SL%(I%)=L%                     ! STARTING BLOCK NO.
1200   UY%(I%)=0                      ! INITIALIZE UNUSED BYTE ACCUMULATOR
1210 !
1220     FOR J%=1% TO SZ%(I%) STEP NY%(I%)   ! ELEMENT COUNTER
1230       IF Y%+NY%(I%)<=512% GOTO 1280     ! BLOCK SIZE EXCEEDED
1240       UY%(I%)=UY%(I%)+512%-Y%           ! ACCUMULATE UNUSED BYTES
1250       Y%=0                              ! INITIALIZE BYTE COUNTER
1260       L%=L%+1%                          ! UPDATE BLOCK NO.
1270       GOTO 1230                         ! START FILLING THE NEW BLOCK
1280       Y%=Y%+NY%(I%)                     ! INCREMENT AN ELEMENT OF BYTES
1290     NEXT J%                             ! GET NEXT ELEMENT
1300 !
1310   EY%(I%)=Y%                  ! ENDING BYTE NO.
1320   EL%(I%)=L%                  ! ENDING BLOCK NO.
1330   UB%(K%)=UB%(K%)+UY%(I%)     ! ACCUMULATE TOTAL UNUSED BYTES
1340   TY%=TY%+SZ%(I%)             ! ACCUMULATE MINIMUM TOTAL BYTES REQUIRED
1350 !
1360 !                        ---- DISPLAY RESULTS ----
1370 !
1380   PRINT #1,USING Y$,TAB(5);MID(SQ$(K%),I%,1%);TAB(17);SY%(I%);
1390   PRINT #1,USING Y$,TAB(29);SL%(I%);TAB(44);EY%(I%);
1400   PRINT #1,USING Y$,TAB(56);EL%(I%);TAB(69);UY%(I%)
1410 !
1420 NEXT I%                ! GET NEXT ARRAY
1430 !
```

```
2000 !********************* DISPLAY TOTALS ***************************
2010 !
2020 AL=EL%(N%)-1%+EY%(N%)/512
2030 TL=TY%/512
2040 AY%=AL*512
2050 PRINT #1, USING Y$,TAB(49);"TOTAL UNUSED BYTES";TAB(69);UB%(K%)
2060 PRINT #1
2070 PRINT #1, USING Y$,TAB(4);N%;" ARRAYS REQUIRING ";TY%;" BYTES IN ";
2080 PRINT #1, USING L$,TL;" BLOCKS"
2090 PRINT #1, USING Y$,TAB(14);"ACTUALLY USE ";AY%;" BYTES IN ";
2100 PRINT #1, USING L$,AL;" BLOCKS"
2110 PRINT #1
2120 TY%=0
2140 !
2150 !            ---- GET SUBSCRIPTS FOR NEXT SEQUENCE ----
2160 !
2170 NEXT C%
2180 NEXT B%
2190 NEXT A%
2200 !
3000 !***************** PICK BEST CHOICE ********************
3010 !
3020 MN%=UB%(1%) \ BC$=SQ$(1%)
3030 FOR K%=1% TO NF%
3040   IF UB%(K%)<MN% THEN MN%=UB%(K%) \ BC$=SQ$(K%) \ K1%=K%  ! PICK SMALLEST
3050 NEXT K%
3060 PRINT #1
3070 PRINT #1,TAB(25);"BEST CHOICE IS TO ORDER THE ARRAYS: ";BC$
3080 PRINT #1,TAB(25);"WHICH YIELDS";MN%;"UNUSED BYTES"
3090 PRINT #1
3100 FOR K%=K1%+1% TO NF%
3110   IF UB%(K%)=MN% THEN PRINT "OR ";SQ$(K%)
3120 NEXT K%
3130 CLOSE 1
3140 END
```

Appendix K
Graphics

INTRODUCTION

The Instrument Controller display offers several useful features that enable the design of customized operator message outputs. These features include character graphics, graphics routines, double-size characters, reverse video, double intensity, and blinking. In addition, the cursor control functions and erasing capabilities give the programmer complete control over the display. With the exception of the graphics routines, these features are all discussed in Section 14 of this manual set.

OVERVIEW

This appendix describes the graphics routines contained in the object file:

GRAPH.OBJ

These routines may be used to create highly formatted displays. The routines included allow plotting and drawing lines, relative to a point or from absolute coordinates. Other routines allow placing a single dot at a specified pixel location, moving (relative and absolute), erasing, and disabling the graphics display.

Section 8 of the 1722A System Guide contains an excellent introduction to display graphics. Read this section first, to familiarize yourself with the terms and concepts used.

GRAPHICS ROUTINES

Information to be displayed is held in two separate portions of memory: the character plane and the graphics plane. Both of them can be turned on and off using the ANSI Standard Mode Selections. Additionally, the graphics plane can be turned on and off using two of the routines in the Object File named "GRAPH.OBJ".

To use the Graphics Routines, the program must link to the object file prior to any command using the graphics plane. Your program must have the line:

```
LINK "GRAPH"
```

before any graphics routines are executed. If you place the line early in any program that will use the graphics display, it will save time by not accessing the disk during program execution to load the graphics routines into memory.

Display Graphics Commands

The paragraphs that follow describe each of the graphics routines. Note that all the arguments must be integers; i.e., they must be followed by the symbol % in any BASIC language program that uses them.

Summary of Commmands

The table that follows describes each of the graphics routines. Note that all the arguments must be integers except for the first arguments to LABEL and LABELF. This means that in a BASIC language program, for example, the arguments must be followed by the % symbol. In FORTRAN, variable types are determined by the first character in the variable name (Integers I though N), or by using the TYPE statement. See the particular progamming language manual for full details.

All of the following routines can be called by a BASIC or FORTRAN program except for LABEL and LABELF. LABEL is called by a BASIC program and LABELF is called by a FORTRAN program.

SUMMARY OF GRAPHICS ROUTINES

COMMAND	PURPOSE
DOT (X, Y, {type})	Draws a single dot at X, Y, returns to the current position.
DRAW (X1, Y1, X2, Y2, {type})	Draws a line from X1, Y1 to X2, Y2.
ERAGRP ({type})	Erases the entire graphics plane.
GRPOFF	Disables the graphics plane.
GRPON	Enables the graphics plane.
MOVE (X, Y)	Moves to absolute position X, Y.
MOVER (Xoffset, Yoffset)	Moves relative to amount specified by the offset.
PAN (X, Y)	Sets display window position to X, Y.
PLOT (X, Y,{type})	Plots from current position to X, Y.
PLOTR (Xoffset, Yoffset, {type})	Plots relative to amount specified by the offset.
LABEL(S$, D%, T%) LABELF(STR,LENSTR,DIR,TYPE, ERROR)	Place string in graphics memory either horizontal or vertical

*Type: -1 = INVERSE
0 = BLACK
1 = WHITE

- The values of the X and Y arguments may not exceed 2047, and may only be negative in the relative commands MOVER and PLOTR.

- The X argument is the number of pixels in the horizontal direction. Since the screen is 640 pixels wide, the center is 320 pixels from the left edge.

- The Y argument is the number of pixels in the vertical direction. The screen is 224 pixels high, so the center is 112 pixels from the bottom edge.

- The 'Type' argument determines whether the routine paints white on black (1), black onto white (0), or the inverse of the color already at that position (–1).

- The S$ argument is a string variable or string constant that is placed in the graphics plane.

- The STR argument is an array with one character in each byte that is placed in the graphics plane.

- The LENSTR argument specifies how many characters are in the STR array.

- The D% and DIR arguments specify whether the string should be placed in the graphics plane horizontally (0) or vertically (90).

In the pages that follow, each of the routines except LABELF is described in detail, and some suggestions are given about the kinds of things that each of them can be used for. Some of the descriptions include program examples. The example listings are all shown as they would appear in programs written for the BASIC Interpreter.

DOT
GRAPH.OBJ

Usage

DOT(X%, Y%, {type%})

Description

This routine places a single dot at the specified coordinates, then returns to the current pixel position. Because this routine returns to the former pixel position it is useful in the construction of detailed charts or graphs that require pixel resolution and are generated by a mathematical formula that calculates each point from the same position.

Parameters

X — An integer 0 to 2047 that specifies the absolute horizontal pixel location for the dot.

Y — An integer 0 to 255 that specifies the absolute vertical pixel location for the dot.

type — An integer value that specifies whether the dot is painted white on black (1), black on white (0), or the inverse of the color already at that position (–1).

Example

This BASIC program uses DOT to draw a sine wave. It first asks for "Amplitude", and then for "Period". The amplitude is the peak-to-peak pixel amplitude of the sine wave that will be drawn. The period is used for frequency and sampling rate. Notice that the program does not allow a period of less than 1. Selecting periods less than five results in waveforms whose resolution is too coarse for the wave to be observable.

```
10  LINK "GRAPH"                        ! link to graphics
20  ERAGRP (0%) \ GRPON                 ! erase, turn on graphics
30  PRINT CHR$(27) + "[2J"              ! clear character plane
40  PRINT CPOS (14,0);"Amplitude";      ! input amplitude
50  INPUT A                             ! amplitude: dots peak-to-peak
60  PRINT CPOS (15,0);"Period",         ! input period (sampling rate)
70  INPUT P                             ! period: no. of dots/cycle
80  IF P = 0 THEN 60                    ! P = 0 is illegal
90  PRINT CHR$(27) + "[2J"              ! clear character plane
100  FOR X% = 1 to 640                  ! x across display
110    Y  = A * SIN (X% / P)            ! calculate y
120    Y% = 0.5 * (Y) + 112             ! integerize and offset y
130    DOT (X%, Y%, 1%)                 ! dot at calc'ed position
140  NEXT X%                            ! continue calculation
150  GOTO 40                            ! next value
```

Usage

DRAW(X1%, Y1%, X2%, Y2%, {type%})

Description

This routine draws a line from absolute position X1, Y1 to another absolute location, X2, Y2. If the final position is beyond the edge of the graphics plane, the line will end at the edge. The current pixel position is not changed by DRAW.

Parameters

X1 An integer 0 to 2047 that specifies the absolute horizontal pixel location for the start of the line.

Y1 An integer 0 to 255 that specifies the absolute vertical pixel location for the start of the line.

X2 An integer 0 to 2047 that specifies the absolute horizontal pixel location for the end of the line.

Y2 An integer 0 to 255 that specifies the absolute vertical pixel location for the end of the line.

type An integer value that specifies whether the line is painted white on black (1), black on white (0), or the inverse of the color already at that position (-1).

Example

Current position is 0,0. To draw a white diagonal line across the display, use:

```
DRAW(0%, 0%, 639%, 223%, 1%)
```

ERAGRP
GRAPH.OBJ

Usage

ERAGRP {type%}

Description

This routine erases the entire graphics plane to the color indicated by {type%}, either green, black, or the reverse of the color before erasing. Any data within the plane will be deleted. The character plane is unaffected.

Parameters

type An integer value that specifies whether the screen is erased with white (1), black (0), or the inverse of the color already at every position in the graphics plane (-1).

Example

At the beginning of a program, use ERAGRP to prepare the Graphics plane for the display.

```
ERAGRP (0%)          ! Erase to black
ERAGRP (1%)          ! Erase to green
ERAGRP (-1%)         ! Create inverse image
```

GRPOFF, GRPON
GRAPH.OBJ

Usage

GRPOFF,GRPON

Description

These routines turn the graphics portion of a display off and on. The memory is left intact; the routines only determine if the graphics plane is displayed or not. The character plane is unaffected.

Example

A selection display has just been presented to the operator. When the selection has been made, a new display is presented that contains new graphics. Rather than using ERAGRP to erase the graphics plane, however, it is desired to leave the contents alone because the test results update the display for the next selection. In this case, use GRPOFF to turn off the graphics display. When the display is updated, the program uses GRPON to display the change.

MOVE
GRAPH.OBJ

Usage

MOVE(X%, Y%)

Description

This routine moves the current pixel location without drawing. If either X% or Y% are outside of the graphics plane, the move stops at the corresponding edge.

Parameters

X — An integer 0 to 2047 that specifies the absolute horizontal pixel location to move to.

Y — An integer 0 to 255 that specifies the absolute vertical pixel location to move to.

Example

A program has just drawn a diagonal line from the bottom left to the upper right corner of the screen. Now, to "lift the pencil" to get back to 0,0, use the MOVE routine:

```
MOVE(0%, 0%)
```

Usage

MOVER(Xoffset%, Yoffset%)

Description

This is the relative move routine. It moves the current pixel position to a relative position within the graphics plane. The move is done without drawing; if the new position is outside the graphics plane, the move stops at the corresponding edge.

Parameters

Xoffset — An integer -2047 to 2047 that specifies the relative number of horizontal pixel locations to move.

Yoffset — An integer -255 to 255 that specifies the relative number of vertical pixel locations to move.

Example

A program is being designed that draws two figures that may appear any place on the display. The second figure must appear immediately to the right of the first. After the first figure is drawn, use the relative move routine to move the current position relative to the ending location of the first figure.

PAN
GRAPH.OBJ

Usage

PAN(X%, Y%)

Description

The PAN routine moves the window around the graphics workspace. The reference is the lower left corner of the display window. Positive arguments move the reference corner to the right and up. Negative arguments move the reference corner left and down. PAN does not affect the current pixel position.

Parameters

X An integer -2047 to 2047 that specifies the relative number of horizontal pixel locations to move the reference corner.

Y An integer -255 to 255 that specifies the relative number of vertical pixel locations to move the reference corner.

Example

During a measurement session, data has been collected by a program, and has become part of a data file. The operator then elects to view the results of the day. The program inserts the raw data into a subroutine that creates and draws a chart that cannot fit in one window. Use the PAN routine to permit viewing the entire chart. Left and right arrow keys can be made part of the display, to allow positioning the window at any area of interest.

Usage

PLOT(X%, Y%, {type%})

Description

This routine draws a line from the current position to the location indicated by the X and Y arguments. (Also see DRAW.) PLOT uses the current position as the starting place to begin drawing, rather than defining the starting position, as DRAW does. The current pixel position is updated to X,Y.

Parameters

X — An integer 0 to 2047 that specifies the absolute horizontal pixel location for the end of the line.

Y — An integer 0 to 255 that specifies the absolute vertical pixel location for the end of the line.

type — An integer value that specifies whether the line is painted white on black (1), black on white (0), or the inverse of the color already at that position (–1).

Example

Use PLOT rather than DRAW in those instances where the starting position will be unknown, but a line is desired from one place to some other position. This routine can be used in constructing some types of graphs, like pie-charts. As the program collects data, the value of the data would be inserted into a Relative Move statement, and the PLOT statement would draw the line from the starting point to the calculated position (which then becomes the new current position).

PLOTR
GRAPH.OBJ

Usage

PLOTR(Xoffset%, Yoffset%, {type%})

Description

The relative plot routine draws a line from the current position to the location indicated by the Xoffset and Yoffset arguments; it is similar to DRAW, except that as it returns to the starting position, continues drawing; it doesn't "lift the pencil".

Parameters

Xoffset — An integer -2047 to 2047 that specifies the relative number of horizontal pixel locations for the end of the line.

Yoffset — An integer -255 to 255 that specifies the relative number of vertical pixel locations for the end of the line.

type — An integer value that specifies whether the line is painted white on black (1), black on white (0), or the inverse of the color already at that position (-1).

Example

A triangular figure is to be drawn, and it may appear anywhere within the graphics plane. Use the Plot Relative routine to draw the figure relative to any starting position. This example draws a triangle that will be black if the field is green, and green if the surroundings are black:

```
MOVE (0%, 0%)
PLOTR(60%, 60%, -1%)
PLOTR(60%, -60%, -1%)
PLOTR(-120%, 0%, -1%)
```

Usage

LABEL({string}, {direction}, {type})

Description

This routine places a string of characters in the graphics plane. These characters appear exactly as they would if displayed normal size in the character plane. The character string can be positioned horizontally or vertically. The current pixel position determines the starting position of the string. The current pixel position is updated so that a subsequent LABEL routine call will cause string concatenation. See subsequent examples on string positioning. If the final position of the string is beyond the edge of graphics memory, the string will wrap around and continue at the opposite edge.

The LABEL routine puts a string into the graphics plane at the rate of about 1/60 of a second per character which means that it takes about 1.3 seconds to draw an 80-character string.

If keyboard input occurs during a long series of drawing operations, there could be up to a 6.5 second delay in responding to the input. This delay time also occurs with ⟨CTRL⟩/C, ⟨CTRL⟩/P, and the ABORT.

Parameters

string — This is a string constant or a string variable. Each character of the string is taken from the character EPROM. If access to the alternate character set is desired, the string should contain the ASCII control character SO (shift out, decimal 14). All characters after SO are taken from the alternate character set until the ASCII control character SI (shift in, decimal 15) is encountered. The SI causes selection to revert to the primary character set. The default is the primary set, so all selection is from the primary set unless SO is encountered.

The maximum length of a string that can be placed in the graphics plane with a single LABEL call is 80 characters. Since the control characters SO and SI are not displayed, they are not part of the 80 character string length. If the 80 character string length is exceeded the string will be truncated.

direction — This is an integer 0 or 90. If the direction is 0, then the string will be horizontally oriented in the graphics plane. If a direction of 90 is given, the string will be positioned vertically. See subsequent examples.

type — This is an integer 0, 1, or –1. It determines whether the routine paints a white label on a black screen (1), black label on a white background (0), or the inverse of the color already at that position (–1).

Errors

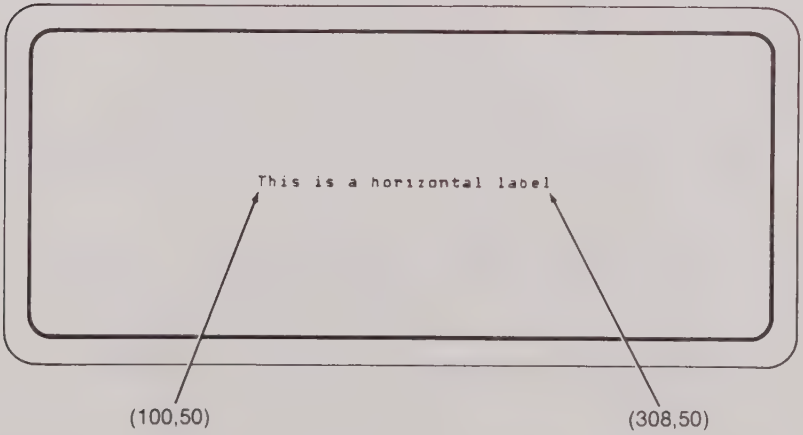
2006 - direction argument not 0 or 90

Example 1

This BASIC program places a 26 character horizontal string in the graphics plane. An initial MOVE changes the current position to 100,150, the place where the bottom left corner dot of the first character cell of the string will be. LABEL updates the current position to the dot immediately to the right of the bottom right corner of the last character cell of the string (308,50). A horizontal character cell is 8 dots wide and 14 dots high, so in this example the current position has been updated in the horizontal direction by 208 dots (26 characters X 8 dots).

```
10 LINK 'graph' \ LINK 'xgraph'          ! link to graphic routines
20 PRINT CHR$(27);"[2J"                   ! clear character plane
30 ERAGRP(0%) \ GRPON \ PAN(0%,0%)        ! erase & turn on graphics plane
40 MOVE(100%,50%)
50 LABEL('This is a horizontal label',0%,1%)
```

The following is what appears on the display screen after the above program is run.



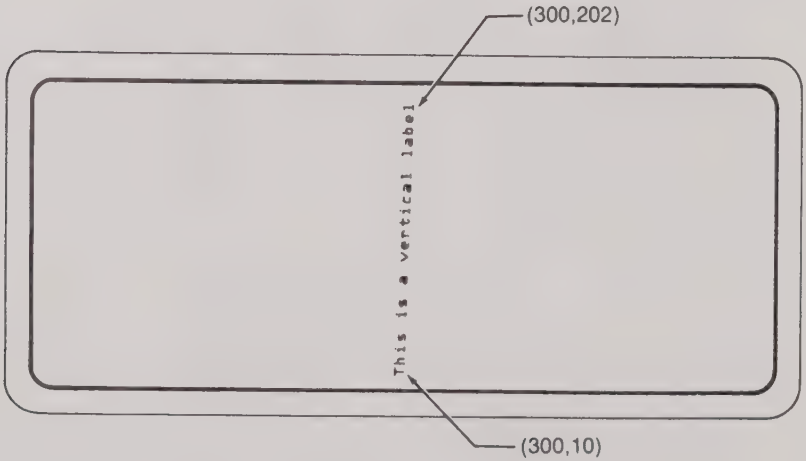
LABEL
GRAPH.OBJ

Example 2

This BASIC program places a 24 character vertical string in the graphics plane. An initial MOVE changes the current position to 300,10, the place where the bottom right corner dot of the first character cell of the string will be. LABEL updates the current position to the dot immediately above the top right corner of the last character cell of the string (300,202). A vertical character cell is 14 dots wide and 8 dots high, so in this example the current position has been updated in the vertical direction by 192 dots (24 characters X 8 dots).

```
10 LINK 'graph'                          ! link to graphic routines
20 PRINT CHR$(27);"[2J"                  ! clear character plane
30 ERAGRP(0%) \ GRPON \ PAN(0%,0%)       ! erase & turn on graphics plane
40 MOVE(300%,10%)
50 LABEL('This is a vertical label',90%,1%)
```

The following is what appears on the display screen after the above program is run.

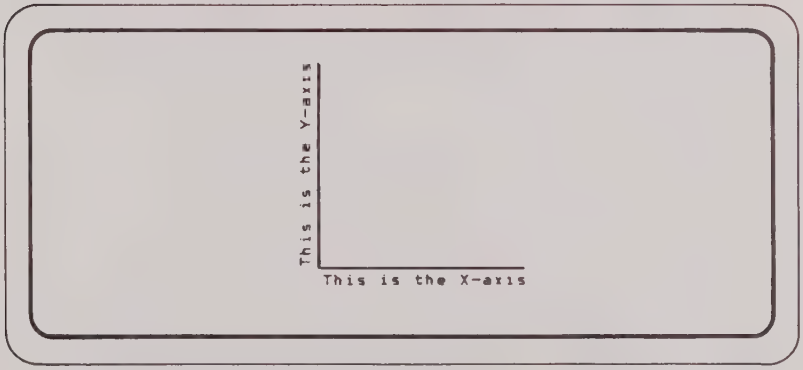


Example 3

This BASIC program draws an X and Y axis with horizontal and vertical labels on the axis. The MOVER and PLOTR commands are used to show how the LABEL routine updates the current pixel position.

```
10 LINK 'graph'                              ! link to graphic routines
20 PRINT CHR$(27);"[2J"                      ! clear character plane
30 ERAGRP(0%) \ GRPON \ PAN(0%,0%)           ! erase & turn on graphics plane
40 MOVE(250%,30%)
50 LABEL('This is the X-axis',0%,1%)
60 MOVER(0%,19%) \ PLOTR(-144%,0%,1%)
70 MOVER(-5%,0%)
80 LABEL('This is the Y-axis',90%,1%)
90 MOVER(5%,0%) \ PLOTR(0%,-144%,1%)
```

The following is what appears on the display screen after the above program is run.



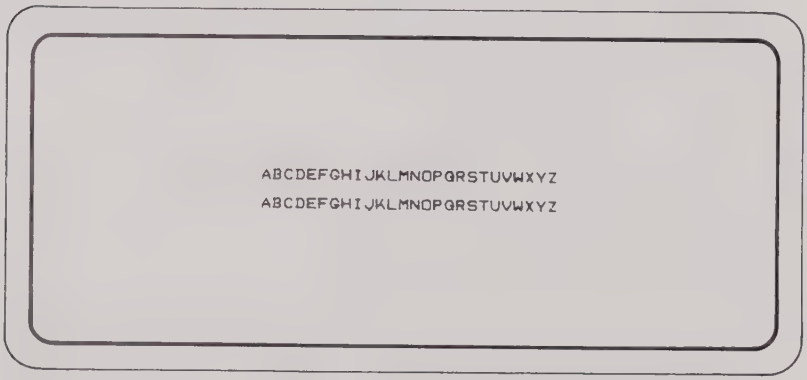
LABEL
GRAPH.OBJ

Example 4

This BASIC program shows how the label command causes the current pixel position to be updated so that strings can easily be concatenated.

```
10 LINK 'graph'                          ! link to graphic routines
20 PRINT CHR$(27);"[2J"                  ! clear character plane
30 ERAGRP(0%) \ GRPON \ PAN(0%,0%)       ! erase & turn on graphics plane
40 MOVE(200%,120%)
50 LABEL('ABCDEFGHIJKLM',0%,1%)          ! some labeling to show concatenation
60 LABEL('NOPQRSTUVWXYZ',0%,1%)
70 MOVE(200%,100%)
80 LABEL('ABCDEFGHIJKLMNOPQRSTUVWXYZ',0%,1%)
```

The following is what appears on the display screen after the above program is run.

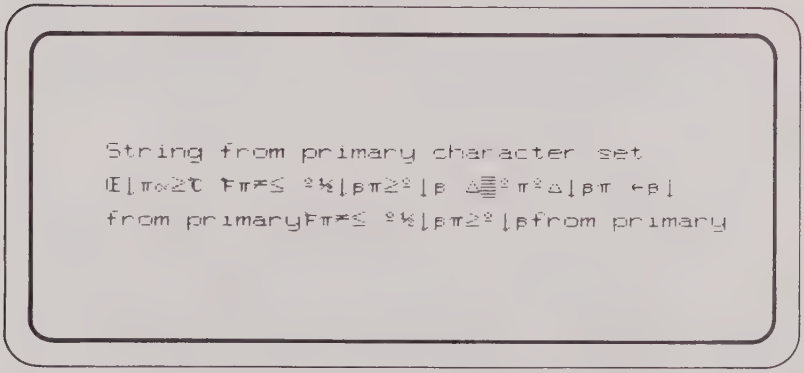


Example 5

This BASIC program shows how the LABEL routine can access the alternate character set.

```
10 LINK 'graph'
20 PRINT CHR$(27);"[2J"                    ! clear character plane
30 ERAGRP(0%) \ GRPON \ PAN(0%,0%)         ! erase & turn on graphics plane
40 SO$=CHR$(14%)
50 SI$=CHR$(15%)
60 MOVE(150%,130%)
70 LABEL("String from primary character set",0%,1%)
80 MOVE(150%,110%)
90 LABEL(SO$+"String from alternate character set",0%,1%)
100 MOVE(150%,90%)
110 LABEL("from primary"+SO$+"from alternate"+SI$+"from primary",0%,1%)
```

The following is what appears on the display screen after the above program is run. This output would be different if a customized (non-standard) character EPROM was being used.



PRINTING THE GRAPHICS PLANE

A printed image of the display screen window into the graphics plane can be obtained by calling special routines from a BASIC or FORTRAN program.

A printer must be attached to a 1722A RS-232 port, KB1: or KB2:. The Epson FX® and RX® series, the Epson MX-100® with GRAFTRAX®, and the Tally® Model MT1605E are the supported printers. A routine is provided that allows flexibility in printing a screen on other printers or a special assembly language module can be written to support other printers. See the section "Printing the Graphics Plane on Unsupported Printers".

The Epson printers need to have the Epson Intelligent Serial Interface® card (Epson Cat. No. 8148) installed. The Epson is connected to the KB1: or KB2: port with Fluke printer cable Y1709. The Epson printer and Epson serial interface DIP switches should all be left in their factory default settings. Use the Set Utility program to establish the following parameters:

```
SET
KB1: br 19200 db 8 pb n si e so e to 10
exit
```

Tally® MT1605E printer should be connected to the KB1: or KB2: port with Fluke printer cable Y1709. All DIP switches should be in the OFF position except switch 4 in the I/O panel, which should be ON. Use the set utility program to establish the following parameters:

```
SET
KB1: br 9600 db 8 pb n si e so e to 10
exit
```

It is important to note that only the display screen window into the graphics plane is printed. The character plane is not printed. A convenient way to place characters in the graphics plane is provided by the graphics routine, LABEL.

® Epson FX, RX, Epson MX-100, and GRAFTRAX are Registered Trademarks of Epson America, Inc
®Tally is a Registered Trademark of Mannesmann Tally

Graphics Print Routines

The routines available to BASIC programs are recorded on the System disk in a file named GPRINT.OBJ, and in a library file named BASIC.LIB. FORTRAN programs access the graphics print routines by using the library file named FLUK22.LIB on the FORTRAN disk.

Summary of Graphics Print Commands

The table that follows describes the graphics print routines. There are two differently named routines for each graphics print capability, one routine is called by a BASIC program and the other is called from FORTRAN. All FORTRAN routine names end in an F; BASIC routine names do not end in an F. All arguments are integers except that the first argument to GRBYTE and GRBYTF is an integer array name.

Summary of Graphics Print Routines

COMMAND	PURPOSE
GPOUT({chan #},{printer type}) GPOUTF({unit #},{printer type},{error})	designate serial port to which to send screen and specify printer type
GPRINT GPRNTF({error})	perform screen dump to printer
GRBYTE({array},{slice},{bit order}) GRBYTF({array},{slice},{order},{error})	read screen slice and store in array

In the pages that follow, each of the BASIC routines is described in detail. Program examples are provided as they would appear in programs written for the BASIC Interpreter. Error numbers given are those that BASIC reports. The corresponding FORTRAN routines are described in the FORTRAN manual.

GPOUT
GPRINT.OBJ

Usage

GPOUT({channel number}, {printer type})

Description

This routine designates a channel for the graphics screen data and the type of printer that is used. This routine must be called before calling the GPRINT routine. The GPRINT routine does the screen dumping and GPOUT is called to prepare for printing.

Parameters

channel number — An integer between 1 and 16 that corresponds to a channel number specified in a previous OPEN statement. The OPEN statement must have assigned a channel to KB1: or KB2:.

printer type — An integer between 0 and n that specifies what type of printer is to be used for the graphics dump.

Printer Type	Printer
0%	Epson FX series
1%	Epson RX series
2%	Epson MX-100 with Graftrax
3%	Tally MT1605E
4%	for user-defined printer types (See Printing the Graphics Plane on Unsupported Printers section)
:	
:	
n%	

Errors

302 - Illegal channel number
308 - Channel is not open
2001 - Undefined printer type

Example

This BASIC program excerpt specifies that the screen to be printed on is an Epson FX printer that is connected to the KB1: port.

```
40 OPEN "KB1:" AS NEW FILE 5          ! assign channel 5 to KB1:
50 GPOUT(5%,0%)                       ! data to channel 5 & Epson FX printer
```

GPRINT
GPRINT.OBJ

Usage

GPRINT

Description

This routine causes the display screen window into the graphics plane to be dumped to a printer. Any images in the character plane are ignored. The BASIC statement after the call to GPRINT is executed when the printing has finished or when an error has been detected.

Almost all input from the keyboard or touch sensitive display is ignored until printing is done or an error is detected. Inputs that are not ignored during the operation of GPRINT are ⟨CTRL⟩/P, ⟨CTRL⟩/C, and the ABORT button. Any of these cause GPRINT to stop, but they have different effects than in a normal BASIC program. These effects are:

- If GPRINT is called from the BASIC Interpreter, ⟨CTRL⟩/C, or the ABORT button will cause processing to resume at the next BASIC statement. If ⟨CTRL⟩/P is done while in character mode (SET NOECHO), control passes to the statement after the GPRINT call. In line mode, ⟨CTRL⟩/P causes an exit to FDOS.

- If GPRINT is called from compiled BASIC while in character mode or line mode, ⟨CTRL⟩/P, ⟨CTRL⟩/C, and the ABORT button will cause an exit to FDOS.

- The BASIC 'ON CTRL/C GOTO ...' statement has no effect during GPRINT execution. This means that while a GPRINT is in operation ⟨CTRL⟩/C, the ABORT button, and possibly ⟨CTRL⟩/P will not be trapped as they normally would be.

Errors

311 - Non-recoverable device read/write error (timeout)
2002 - GPOUT not called yet
2003 - Printer timeout
2004 - KB0: is not the console device (a switch has been performed)

GPRINT
GPRINT.OBJ

Notes

1. Pressing the ABORT button during GPRINT can cause BASIC error 311 (Non-recoverable device read/write error). A ⟨CTRL⟩/S done before or during GPRINT will always cause a 311 error.

2. If ⟨CTRL⟩/P is done during a GPRINT, it may be necessary to power the printer off and on. This is to avoid extraneous data to be printed at the start of a subsequent screen print.

Example

This BASIC program draws a picture and prints it on an Epson MX-100 printer.

```
10 LINK 'gprint' \ LINK 'graph'      ! link to graphic & print routines
20 PRINT CHR$(27);"[2J"              ! clear character plane
30 ERAGRP(0%) \ GRPON                ! erase & turn on graphics plane
40 MOVE(250%,80%)
50 PLOTR(60%, 60%, -1%)              ! draw a picture
60 PLOTR(60%, -60%, -1%)
70 PLOTR(-120%, 0%, -1%)
80'
90 OPEN "KB1:" AS NEW FILE 4         ! assign channel 4 to KB1:
100 GPOUT(4%,2%)                     ! data to channel 4 & an Epson MX-100
110 GPRINT                           ! print graphics plane
120 CLOSE 4
```

GRBYTE
GPRINT.OBJ

Usage

GRBYTE({array address}, {slice number}, {slice row order})

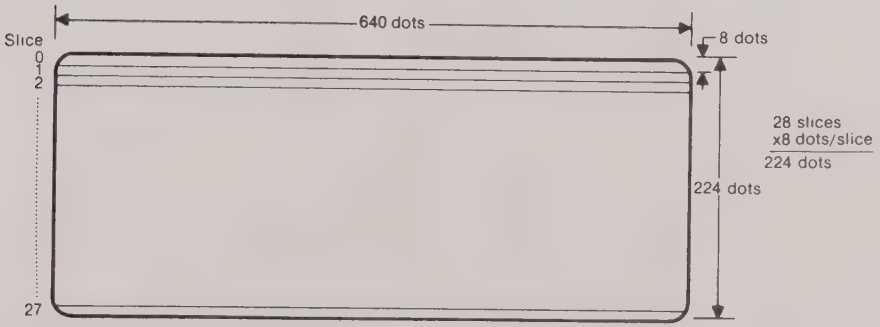
Description

This routine fills an array with a "slice" of the display screen. Any images residing in the character plane are ignored. This routine can be used for printers not supported by the GPRINT routine, or for saving a screen image file to print later.

Parameters

array address — This is the starting address of a 640 element integer array. This array must be a main memory array; it can not be a virtual array. The first address of the array is specified by the array name followed by left and right parentheses, for example A%(). The program can fail if the array does not contain at least 640 elements.

slice number — An integer 0 through 27 which corresponds to an eight dot high horizontal section of the display screen. The following figure shows what is meant by a "slice" of the screen and how the slices are numbered.

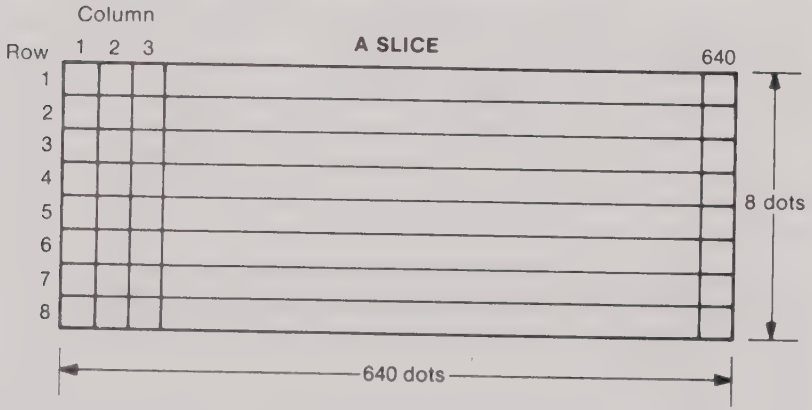


slice row order

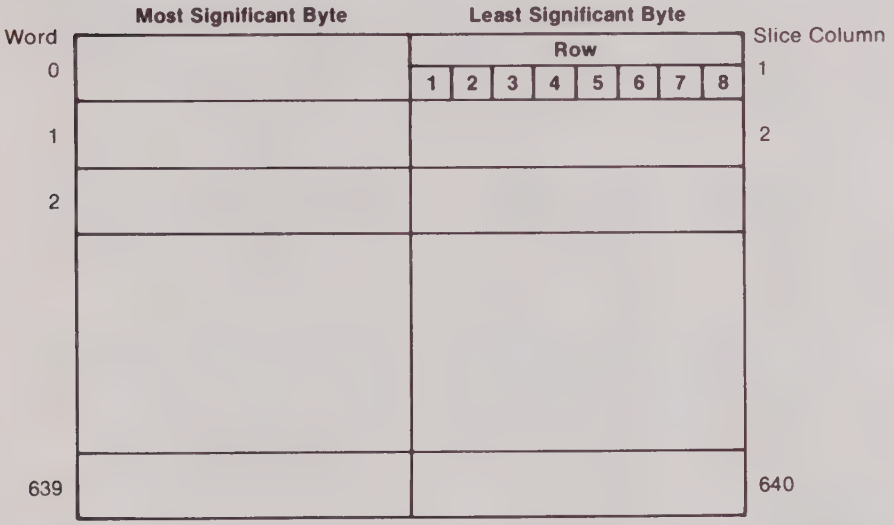
An integer that specifies the order in which the bits in each column of a slice are placed into the array. If this argument is 0, the bottom row of a slice is placed in the least significant bit of an array element. If the argument is any other value, the top row of a slice is placed in the least significant bit of an array element. This argument is provided because of different printer model conventions in addressing the print head pins.

The GRBYTE routine takes a slice of the display screen and places the information in the least significant bytes of the user's array. If a dot location is on (illuminated), then the bit in the array corresponding to that dot location is a 1. The following figures show how a slice of the screen is placed into the user's array when the 'slice row order' argument is 0.

SLICE TO ARRAY CORRESPONDENCE

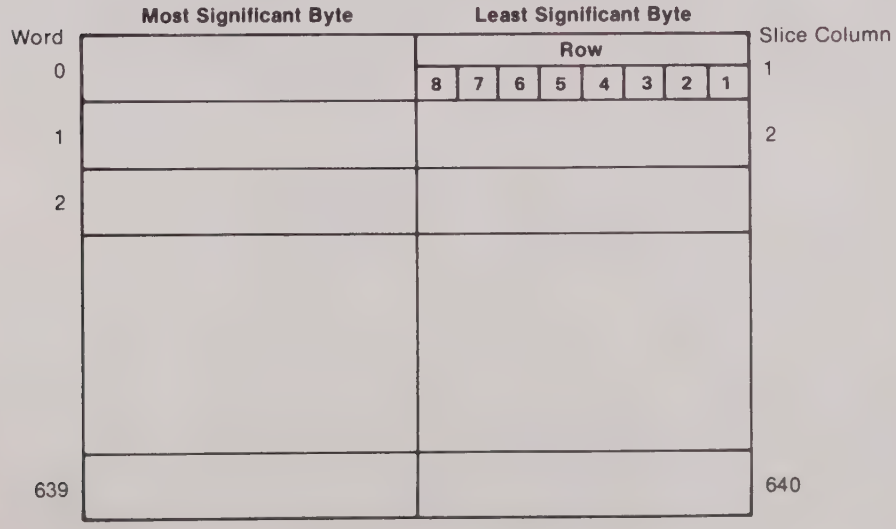


USER ARRAY WHEN SLICE ROW ORDER ARGUMENT IS 0



The following figure shows how a slice of the screen is entered into the user's array when the "slice row order" argument is not 0.

USER ARRAY WHEN SLICE ROW ORDER ARGUMENT IS NOT 0



GRBYTE
GPRINT.OBJ

Errors

311 - Non-recoverable device read/write error (timeout)
2004 - KB0: is not the console device (a switch has been performed)
2005 - Slice number not in 0 to 27 range

Note

Pressing the ABORT button during GRBYTE can sometimes cause BASIC error 311 (Non-recoverable device read/write error). A ⟨CTRL⟩/S done before or during GRBYTE will always cause a 311 error.

Example

This BASIC program draws a picture and prints it on an Epson FX printer using an IEEE port. The program assumes that the Epson's IEEE interface card has been set to device address 6.

```
10 LINK 'gprint' \ LINK 'graph'       ! link to graphic & print routines
20 ERAGRP(0%) \ GRPON \ PAN(0%,0%)
30 MOVE(0%,0%)
40 PLOTR(0%,223%,1%)                  ! draw a picture
50 PLOTR(639%,0%,1%)
60 PLOTR(0%,-223%,1%)
70 PLOTR(-639%,0%,1%)
80 DIM A%(640%)                       ! array for a screen 'slice'
90 PRINT @6,CHR$(27);'A';CHR$(8);     ! send code for printer line spacing
100 FOR S% = 0% TO 27%
110   GRBYTE(A%(), S%, 0%)            ! a slice of screen to array A
120   PRINT @6,CHR$(27);'*';CHR$(6);  ! send required code to put Epson FX
130   PRINT @6,CHR$(128);CHR$(2);     !       in graphics mode
140   FOR I% = 0% TO 639%
150     PRINT @6,CHR$(A%(I%));        ! send 8 dot column of slice to printer
160   NEXT I%
170  PRINT @6                         ! carriage return
180 NEXT S%
```

PRINTING THE GRAPHICS PLANE ON UNSUPPORTED PRINTERS

The printers supported by the GPOUT and GPRINT routines are the Epson FX and RX, the Epson MX-100 with GRAFTRAX, and the Tally MT1605E. There are two different ways to print the graphics screen data on other printers.

1. The easiest method of printing the screen on various printers is to use the GRBYTE routine. Flexibility to drive different printers is achieved by making multiple calls to the GRBYTE routine from a BASIC program and using the BASIC PRINT statement. The GRBYTE routine retrieves the screen data and the PRINT statement sends the screen data to a printer. See the previous section that describes the GRBYTE routine.

The following is an example of a BASIC program that calls GRBYTE and uses PRINT statements to print a screen image on a Prowriter Model M8510B® printer.

```
10 LINK 'gprint' \ LINK 'graph'     ! link to graphic & print routines
20 ERAGRP(0%) \ GRPON \ PAN(0%,0%)
30 MOVE(0%,0%)
40 PLOTR(0%,223%,1%)                ! draw a picture
50 PLOTR(639%,0%,1%)
60 PLOTR(0%,-223%,1%)
70 PLOTR(-639%,0%,1%)
80 DIM A%(640%)
90 OPEN "KB1:" AS NEW FILE 2        ! assign channel 2 to KB1:
100 PRINT #2,CHR$(27);'T14';        ! send code for printer line spacing
110 FOR S% = 0% TO 27%
120    GRBYTE(A%(), S%, 1%)         ! a slice of screen to array A
130    PRINT #2,CHR$(27);'S00640';  ! put Prowriter M8510B in graphics mode
140    FOR I% = 0% TO 639%
150       PRINT #2,CHR$(A%(I%));    ! send 8 dot column of slice to printer
160    NEXT I%
170  PRINT #2                       ! carriage return
180 NEXT S%
190 CLOSE 1
```

® *Prowriter Model M8510B is a Registered Trademark of C. Itoh Electronics, Inc.*

2. Another method of printing on an unsupported printer is to create an Assembly Language routine and tables modeled after the assembly routines and tables for the supported printers. These printer driver routines and tables are recorded on the Assembly disk (Option 17XXA-201) in a file named PRDRIV.PRE.

   A new printer type value to be specified in the GPOUT routine is defined by supplying a new entry in the file named PTYPE.PRE. The Assembly disk has an image of this Assembly Language source file.

   By writing new routines and tables in PRDRIV.PRE and making the appropriate entry in PTYPE.PRE, a BASIC program only needs to call GPOUT and GPRINT to print the display screen.

   The steps to be taken and excerpts from the above mentioned Assembly Language source are as follows:

   a. Add a new printer type value and entry point name in PTYPE.PRE. The following is the entire source of PTYPE.PRE. Entries would be made before the 'data 0,0' line and after the 'tally' line.

```
*       idt     'ptype'
        def     ptype           ;table of printer types & module names
        ref     epsonf          ;Epson FX & Epson RX driver
        ref     epsonm          ;Epson MX driver
        ref     tally           ;Tally MT1605E
*
*               type module       printer
*               ---- ------       -------
ptype   data    0,epsonf        ;Epson FX
        data    1,epsonf        ;Epson RX
        data    2,epsonm        ;Epson MX
        data    3,tally         ;Tally MT1605E
        data    0,0             ;null to mark end of table
        end
```

b. Add a def statement for the new printer driver entry point at the start of PRDRIV.PRE

c. Add a new module in PRDRIV.PRE that references a new printer dependency table. The following is the module for the Epson FX printer.

```
*
* module:       epsonf
*
* function:     To send graphics screen data to the Epson FX or RX printer
*
* called via:   bl from GPRINT
*               (all registers available except r11)
*
epsonf  equ     $               ;entry point
        clr     r1              ;
*       if      @slicen eqw r1  ;if at 1st screen slice
        li      r1,epfdat       ;    get Epson FX & RX printer dependencies table
        mov     r1,@prdptb      ;
*       endif                   ;endif
        b       @driver         ;go to common printer driver
```

d. Create a printer dependency table in PRDRIV.PRE. Most entries in this table are pointers to other tables that contain the character sequences required by a particular printer. The third entry in this table is the time in 10 millisecond ticks that it takes for the printer to print its data buffer. This entry is required for handling CTRL /C and the ABORT button correctly. The order of entries within this printer dependency table is very important. Use the following table for the Epson FX printer as a guide.

```
*
*          Epson FX & RX dependency table
*
epfdat          data    epstch  ;row start characters table address
                data    epstln  ;row start characters table length
                data    200     ;delay in 10ms ticks for ^C during printing
                data    epterm  ;termination characters table address
                data    epterl  ;termination characters table length
                data    0       ;null to mark end of this table
```

e. Create the tables containing the character sequences that are required to put the printer in dot graphics mode. The following are tables used by the Epson FX printer driver.

```
epstch          byte    27      ;Epson FX & RX row start characters
                byte    'A'     ;
                byte    8       ;'ESC A 8' sets line feed spacing to 8/72"
                byte    27      ;
                byte    '*'     ;
                byte    6       ;
                byte    128     ;
                byte    2       ;'ESC * 6 128 2' graphics (90dpi) 640 dot row
epstln          equ     $-epstch
                even
*
epterm          byte    27      ;Epson FX,RX,MX-100 screen dump term characters
                byte    '@'     ;'ESC @' master reset for Epson (power up state)
epterl          equ     $-epterm
                even
```

f. Assemble the modified PTYPE.PRE and PRDRIV.PRE. Create a new GPRINT.OBJ file by issuing the following Linkage Editor commands. The files PRINT.OBJ and F$RGMY.OBJ that are referenced in these LE commands are also provided on the Assembly disk.

```
le gprint.obj
task gprint
form ascii
part
incl print,prdriv,ptype,f$rgmy
end
```

g. Call the GPOUT routine in a BASIC program with the new printer type value and then call the GPRINT routine.

DISPLAY GRAPHICS PROGRAMMING EXAMPLE

The following program example illustrates using the graphics routines to create a menu for use with the TSO. Multiple-use subroutines are used to minimize program size. Lines 10 through 230 store constants and initialize variables used through the program. These constants include all of the character enhancement control sequences, even though not all of them are used. Once program development is complete, the unused control sequences may be removed.

The subroutine between lines 300 and 360 is used to draw 4 boxes on the graphics plane. The FOR-NEXT loop in the subroutine calls the box plotting subroutine at line 1870 and passes 4 parameters used by the subroutine. These parameters determine the relative X and Y position, the offset on the graphics plane, and the color used to plot the box. The size of the box is fixed by the box plotting subroutine. The boxes are labeled beginning at line 430 with the contents of string array A0$.

At line 480 T% controls the wait time of the screen touch subroutine. If T% = 0, then the wait time is infinite, otherwise the wait time is equal to T% in milliseconds. When a TSO square is touched, the square number is returned in K%.

Line 525 sends the character enhancements for blinking and high-intensity characters to the display module. Line 527 sets T% to 10000, which causes the screen touch subroutine to wait 10 seconds for TSO input before returning.

A series of IF-THEN statements beginning at line 530 decodes the input from the TSO and directs program execution to the proper point in the program. If any of the IF-THEN statements are true, then the subroutine at line 630 is called, otherwise the program loops back to line 300 and re-draws the menu. The subroutine at line 630 prints the non-destructive character (ESC=) over the selected menu entry, causing the selected menu entry to take on the attributes set in line 525.

The plotting subroutine may be used to "undraw" the boxes from the graphics plane. This is useful when other graphics material has been acumulated on the graphics plane and you need to selectively erase the boxes. The subroutine at line 3000 does this by setting F1% to 0 and calling the box drawing subroutine at line 310.

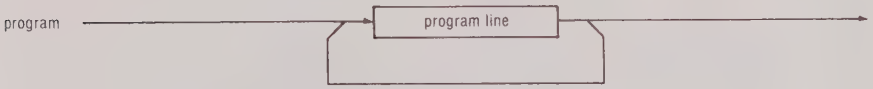
```
10! menu.bas
20!
30!
40!
50 LINK "GRAPH"
60 ON CTRL/C GOTO 1960                  ! ctrl/c handler
110!  constant definitions
120 F1% = 0%
130 DIM A0$(10%)                        ! array for name storage
140 ES$ = CHR$(27)+'['                  ! standard escape sequence
150 CS$ = ES$+"2J"                      ! clear screen
160 FA$ = ES$+"?11" \ CA$ = ES$+"?1h"   ! field \ character attr
170 FL$ = ES$+"5m" \ NA$ = ES$+"m"      ! flash \ normal attr
180 HI$ = ES$ + "1m"                    ! hi intensity
190 NS$ = ES$+"?21" \ DS$ = ES$+"?2h"   ! normal \ double size
200 SU$ = CS$+ES$+'?51'+DS$+ES$+'?81'+ CA$ !cleanup screen
210 ND$ = CHR$(27) + "="                ! on-destructive character
219 ! store names for menu
220 A0$(1) = "run program" \ A0$(3) = "run program #2"
230 A0$(5) = "erase boxes" \ A0$(7) = "exit"
251!
252                                      ! beginning of menu loop
260 PAN(0%,0%)                           !initialize window pos
280 PRINT SU$                            !initialize display
285 ERAGRP(0%)                           !erase the graphics plane
290!                draw four boxes
300 F1% = 1%                            ! fw = 1 TO draw, 0 to erase
310 RL% = 2000% \ XG% = 360%            ! rl = offset, xg = x position
320 FOR I% = 1 TO 4                     ! loop to draw boxes
330 YG% = (I% * 52%)                    ! box y position
340 GOSUB 1870                          ! box draw subroutine
350 NEXT I%
354!                                F1% does double duty as subr flag
355 IF F1% = 0% THEN RETURN
360 GRPON \ PAN (1540%,15%)             ! graphics on, set things up
370!
380!                label them
390 ON CTRL/C GOTO 390 \ OFF KEY        !on ctrl c do menu again
420!                paint menu
425 GRPON
430 PRINT CA$;CPOS(1,0);A0$(1);
440 PRINT CPOS(3,0);A0$(3);
450 PRINT CPOS(5,0);A0$(5);
460 PRINT CPOS(7,0);A0$(7);
480 T% = 0 \ GOSUB 2010                 !get tso key; t is wait time
490 ON CTRL/C GOTO 1960                 !ctrl/c to exit
500 PRINT CHR$(7)                       !to tell them they got a hit
510!
520! decipher which touch sense key it was, make that entry flash
525 PRINT FL$;HI$;                      !flashing, hi intensity
526 !print cpos(15,0);k%;               !for debugging TSO locations
527 T% = 10000
530 IF K% = 9% OR K% = 10% THEN 540 ELSE 550
540 X% = 1% \ GOSUB 630 \ GOSUB 1000    ! selection #1
550 IF K% = 20% OR K% = 30% THEN 560 ELSE 570
560 X% = 3% \ GOSUB 630 \ GOSUB 1530    ! selection #2
570 IF K% = 39% OR K% = 40% THEN 580 ELSE 590
580 X% = 5% \ GOSUB 630 \ GOSUB 3000    ! selection #3
590 IF K% = 59% OR K% = 60% THEN GOTO 600 ELSE 610
600 X% = 7% \ GOSUB 630 \ GOTO 1960     ! exit
610 GOTO 300                            ! invalid key, go back
620 ! subroutine to paint nd character on menu entry
630 PRINT CPOS(X%,0);DUPL$(ND$,LEN(A0$(X%)));
640 WAIT 2000 \ RETURN
1000 GRPOFF \ PRINT CS$;CPOS(1,0);"menu selection #1"; \ GOSUB 2000
1010 K% = 0 \ RETURN
1530 GRPOFF \ PRINT CS$;CPOS(1,0);"menu selection #2"; \ GOSUB 2000
1540 K% = 0 \ RETURN
1860 ! routine to draw boxes
1870 GRPON
1880 BH% = 60% \ BV% = 40% ! box horiz, box vert dimensions
1890 MOVE (XG%+RL%, YG% )  ! move to the right place
1900 PLOTR(BH%,0%,F1%)     ! plot relative to draw the box
1910 PLOTR(0%,-BV%,F1%)
1920 PLOTR(-BH%,0%,F1%)
1930 PLOTR(0%,BV%,F1%)
1940 RETURN
```

```
1948! clean house, go home
1960 OFF CTRL/C \ PRINT CS$; NS$;
1962 GRPOFF \ END
1999 ! touch screen routine
2000 PRINT NA$;CPOS(15,5);"touch screen to continue";
2001 PRINT CHR$(13);          !alternate entry point
2005 K% = KEY                 !zero KEY variable
2010 IF T% <> 0 THEN WAIT T% FOR KEY \ GOTO 2030 !optional wait time
2020 IF T% = 0 THEN WAIT FOR KEY \ GOTO 2030
2030 K% = KEY \ RETURN
3000 ! subroutine to erase boxes
3010 GRPON                    !graphics on
3015 F1% = 0 \ WAIT 1000      !reset the flag, wait a second
3020 GOSUB 310                !go erase them
3030 T% = 20000 \ GOSUB 2000 !wait 20 seconds for touch
3040 K% = 0 \ RETURN          !reset k%, return
3050 END
```

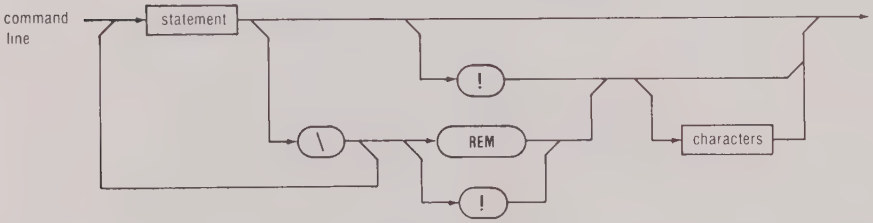
Appendix L
Supplementary Syntax Diagrams

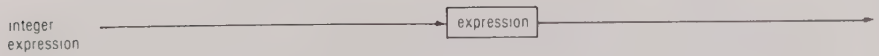
These diagrams supplement those used throughout this manual. They are in three categories: Program and Command Line Syntax, Expression Syntax, and Miscellaneous Syntax. The last category alphabetically lists fundamental syntactic elements at an elementary level.

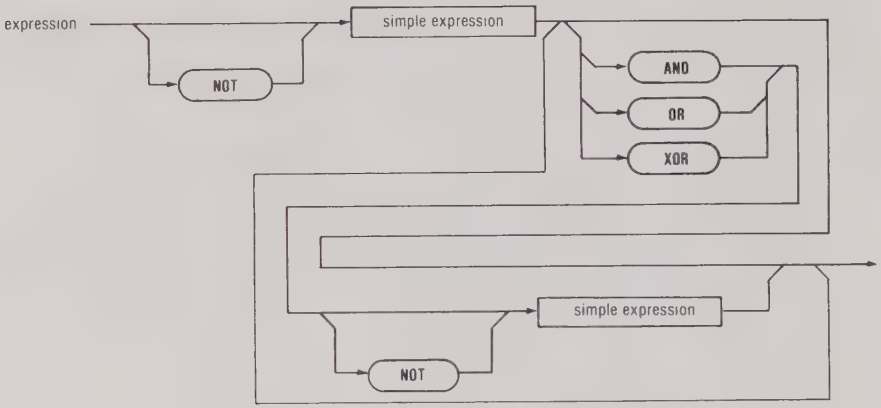
Program and Command Line Syntax

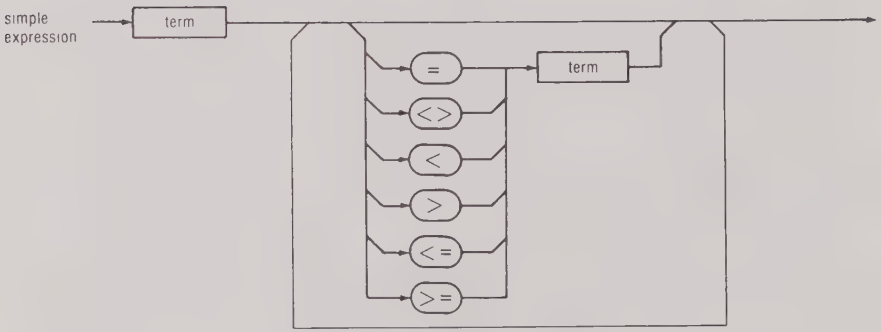


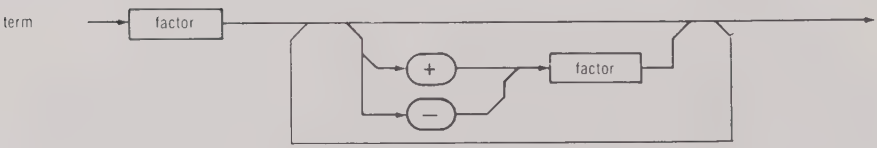
Statement: given in main sections of manual and in Reference.

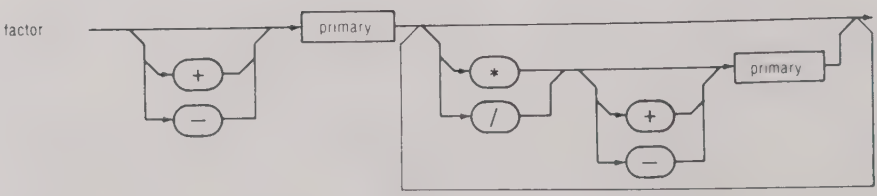


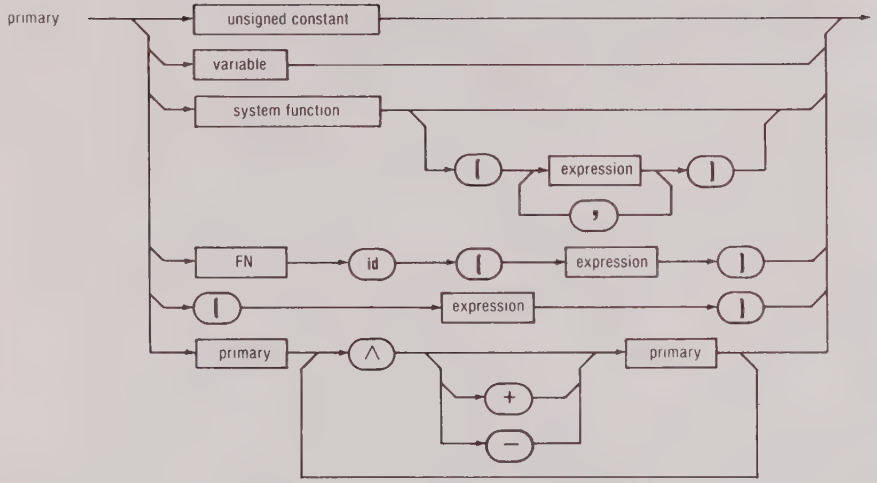



expression
simple expression
NOT
AND
OR
XOR
simple expression
NOT

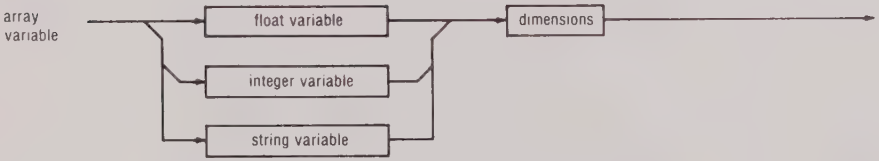

simple expression
term
=
<>
<
>
<=
>=
term

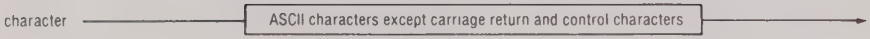

term
factor
+
−
factor

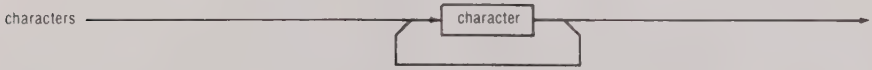

factor
primary
+
−
*
/
primary
+
−

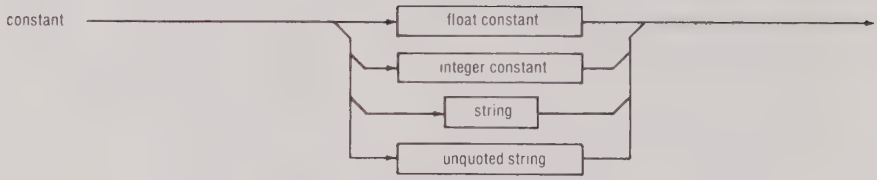


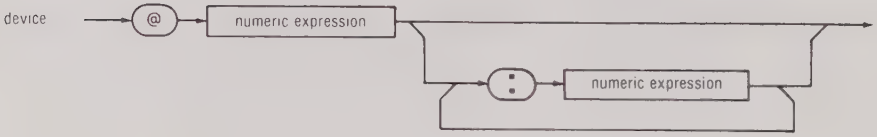
Miscellaneous Syntax

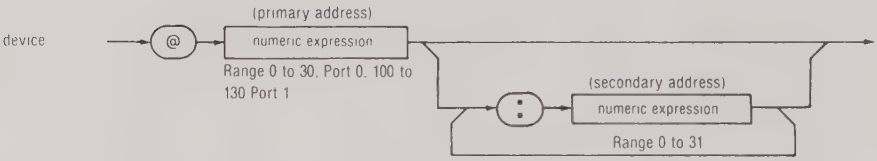


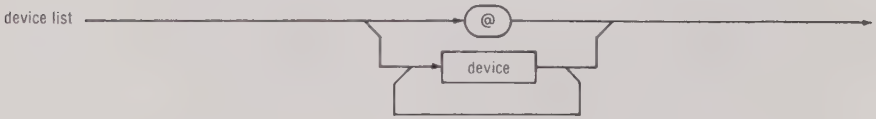


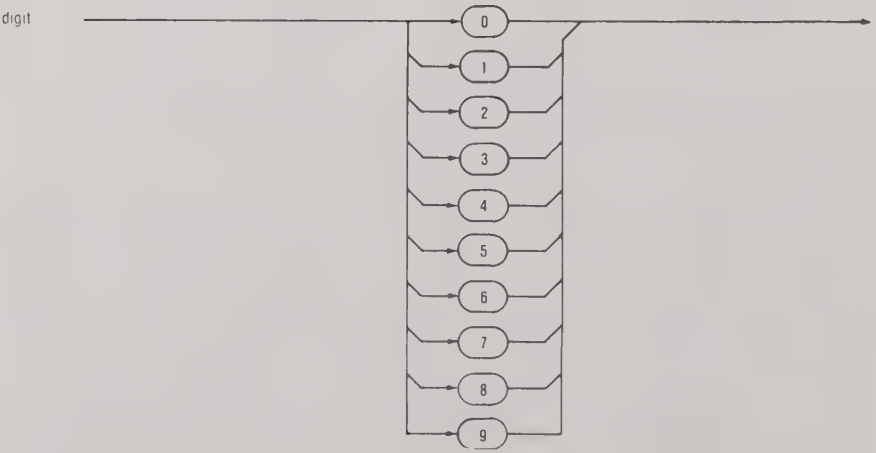


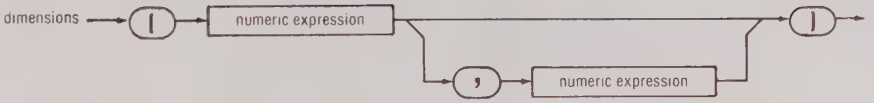

constant
float constant
integer constant
string
unquoted string

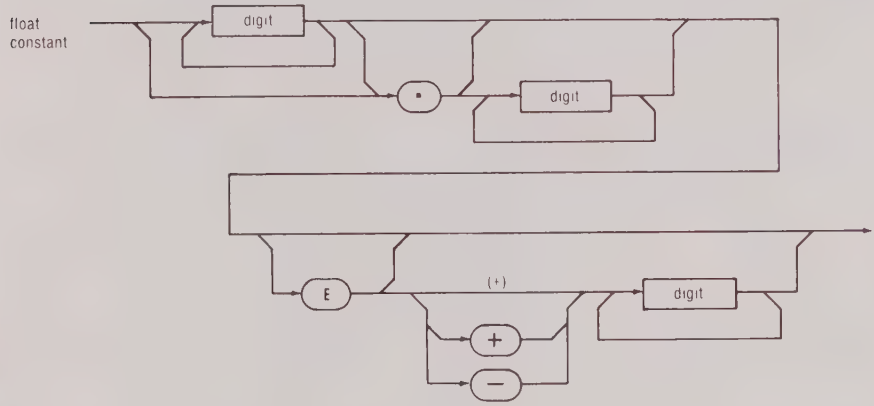

device
@
numeric expression
:
numeric expression

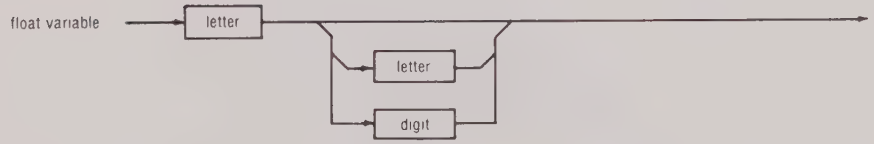

device
(primary address)
@
numeric expression
Range 0 to 30, Port 0, 100 to 130 Port 1
:
(secondary address)
numeric expression
Range 0 to 31

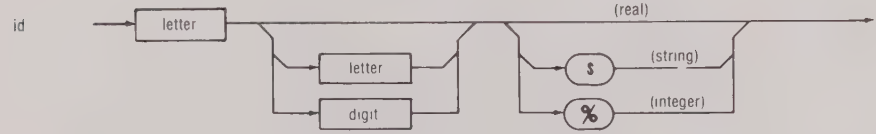

device list
@
device

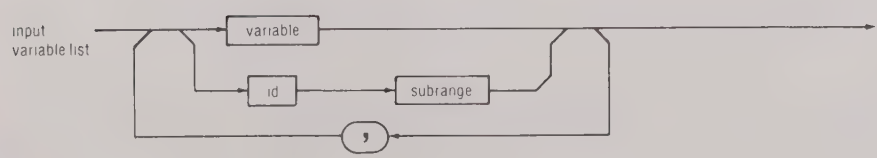

digit
0
1
2
3
4
5
6
7
8
9

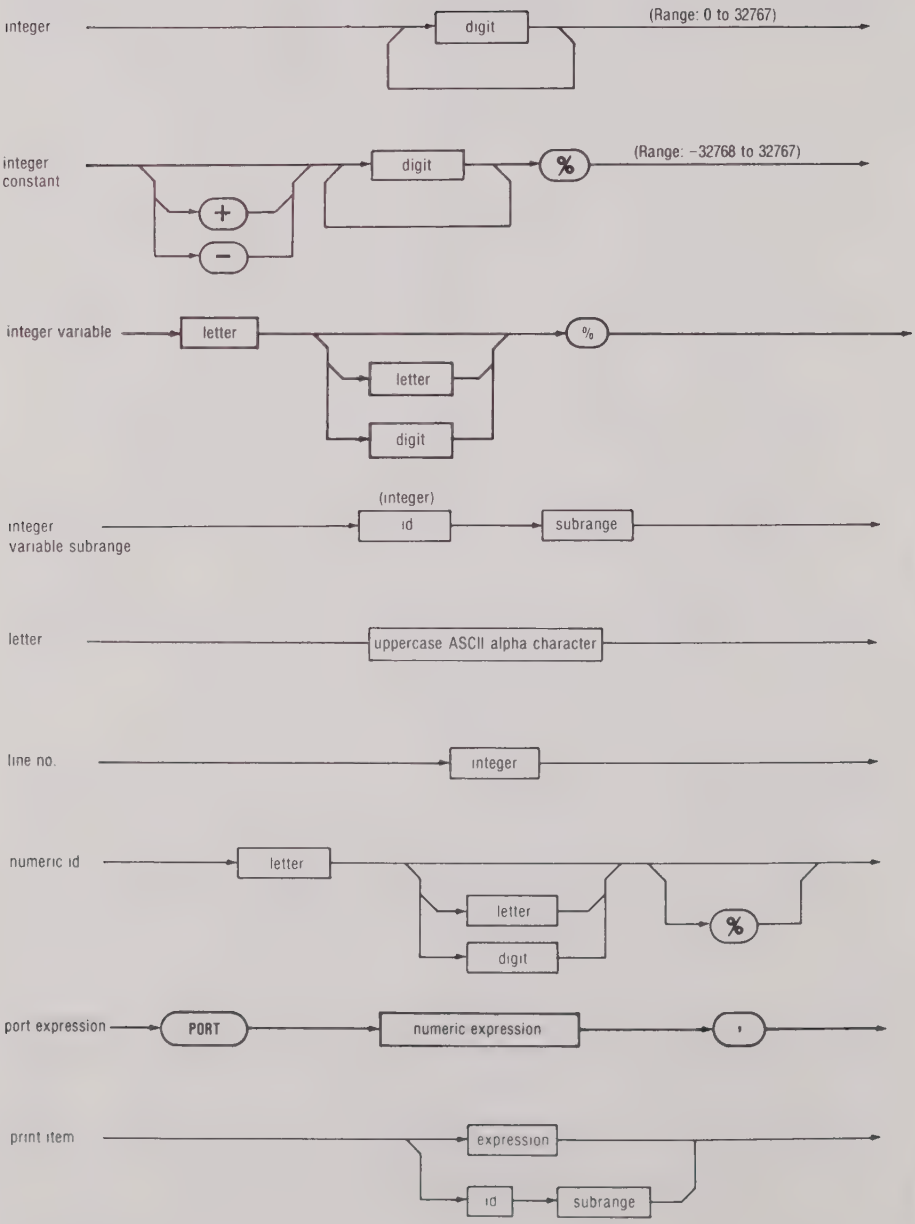

dimensions
(
numeric expression
)
,
numeric expression

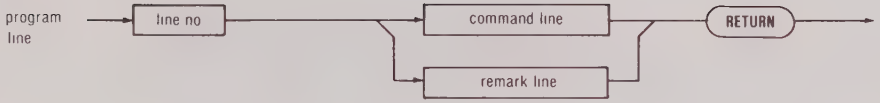

float
constant
digit
.
digit
E
(+)
digit
+
−

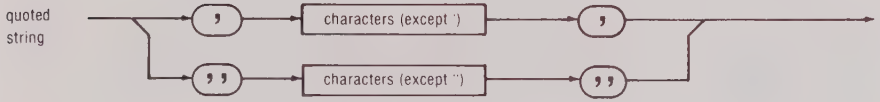

float variable
letter
letter
digit

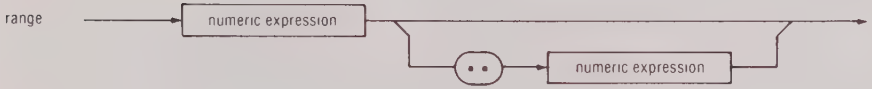

id
letter
letter
digit
(real)
$
(string)
%
(integer)

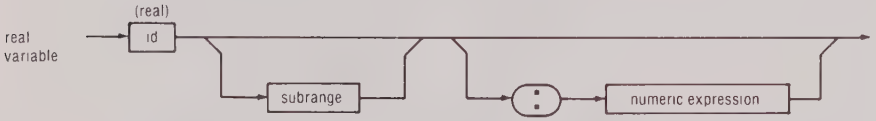

input
variable list
variable
id
subrange
,

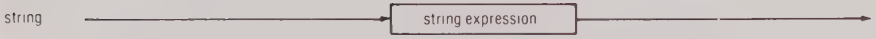

integer
digit
(Range: 0 to 32767)
integer constant
+
-
digit
%
(Range: -32768 to 32767)
integer variable
letter
letter
digit
%
integer variable subrange
(integer)
id
subrange
letter
uppercase ASCII alpha character
line no.
integer
numeric id
letter
letter
digit
%
port expression
PORT
numeric expression
,
print item
expression
id
subrange

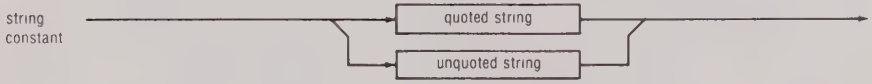

program line
line no
command line
remark line
RETURN

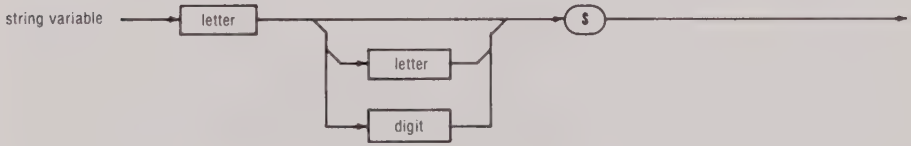

quoted string
'
characters (except ')
''
characters (except ")

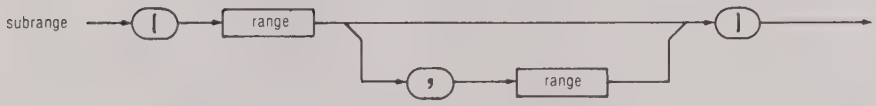

range
numeric expression
..
numeric expression

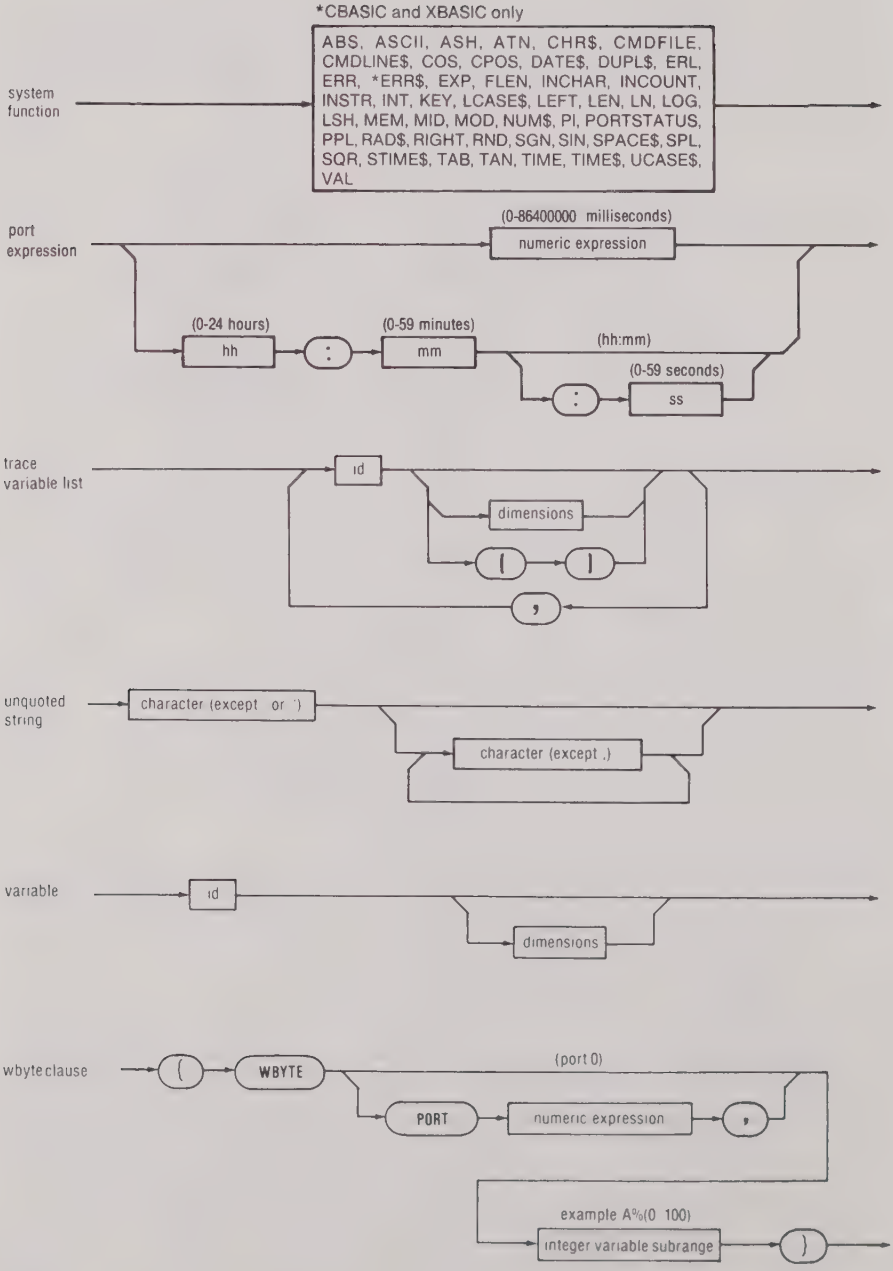

real variable
(real)
id
subrange
:
numeric expression


string
string expression


string constant
quoted string
unquoted string


string variable
letter
$
letter
digit


subrange
(
range
)
,
range


*CBASIC and XBASIC only
system function
ABS, ASCII, ASH, ATN, CHR$, CMDFILE, CMDLINE$, COS, CPOS, DATE$, DUPL$, ERL, ERR, *ERR$, EXP, FLEN, INCHAR, INCOUNT, INSTR, INT, KEY, LCASE$, LEFT, LEN, LN, LOG, LSH, MEM, MID, MOD, NUM$, PI, PORTSTATUS, PPL, RAD$, RIGHT, RND, SGN, SIN, SPACE$, SPL, SQR, STIME$, TAB, TAN, TIME, TIME$, UCASE$, VAL
port expression
(0-86400000 milliseconds)
numeric expression
(0-24 hours)
hh
:
(0-59 minutes)
mm
(hh:mm)
:
(0-59 seconds)
ss
trace variable list
id
dimensions
(
)
,
unquoted string
character (except or ')
character (except ,)
variable
id
dimensions
wbyteclause
(
WBYTE
(port 0)
PORT
numeric expression
,
example A%(0 100)
integer variable subrange
)

Appendix M
Glossary

ABORT

Front panel push switch that causes the Controller to send Device Clear (DEL) and to simulate the entry of CTRL /C. When pressed simultaneously with RESTART, causes the system to perform a cold start.

address

A coded number representing the location of an item. Examples include bus address and program address.

Address command

A bus command from a controller commanding an instrument at a designated address to talk or listen.

Addressed command

A bus command from a controller intended for all instruments that have been addressed as talker or listener.

alias

A shortened or more familiar form of a command. In the 1722A, all aliases must be recorded on a file named ALIAS.SYS which is read by the FDOS program when the system is bootloaded.

application program

A user written program designed to perform specified functions in a working environment.

array

A collection of data items, organized as a row x column matrix.

array element identifier

The subscript of an array variable that identifies the row and column of the desired array element. In the expression: A$(3,5), (3,5) is the array element identifier, referring to row 3, column 5.

ASCII

Acronymn for American Standard Code for Information Interchange. ASCII is a standardized code set of 128 characters, including full alphabetic (upper and lowercase), numerics, and a set of control characters.

asynchronous data

Information transmitted at random times, normally one character at a time, and at predefined, self-clocking baud rates. See synchronous data.

BASIC

Beginner's All-purpose Symbolic Instruction Code, a general purpose, high-level language that has been widely accepted because of its versatility and the ease with which it can be learned. Fluke BASIC has added commands for instrument control.

baud rate

The serial transfer rate in bits per second including all framing bits used to identify the start and end of characters or messages.

binary

A number system based on zero (0) and one (1) representations. It is often used to represent data or instruction codes. There are only two numbers, so digital computers can use binary for their operations because each number can be represented as the state (on or off) of a transistor.

bit

A contraction of binary digit. A bit is either a one or a zero and respresents the smallest single unit of computer information. Bits are often used in groups of eight to represent ASCII characters.

block

Memory size equal to 512 bytes.

bootstrap

A short program permanently recorded in ROM whose only function is to read an operating system program from bulk storage into system memory and transfer control to it.

buffer

A temporary storage area in main memory used to store data.

bulk storage

A device attached to a computer that can store much more program or data information than the computer's main memory can hold. the Instrument Controller incorporates two types of bulk storage: floppy disk and hard disk. Also called mass storage.

bus

The IEEE-488-1980 standardized interconnection system used for connecting instruments. Also, bus can refer to any set of parallel connections that have the same meaning for each unit connected to them.

Bus address

A 7-bit code placed on the IEEE-488 bus in command mode to designate an instrument as a talker or listener.

byte

A grouping of eight bits of information into a coded representation of all or part of a number or instruction. Often a 7-bit ASCII character is referred to as a byte, with the eighth bit available for parity if needed. Bytes are commonly considered as 8-bit storage areas to represent ASCII characters.

chaining

A method of operating a program that is larger than available main memory. The technique is to break the program into smaller elements, and call in the next element from bulk storage as each succeeding element is completed. Requires highly modular programming to be effective. See structured programming.

channel

A communication path opened between an application program and a file or a system device.

character plane

The portion of the display memory that is used for displaying the normal- and double-sized characters. The character set includes the upper- and lower-case alphabet, the ten numerals, and punctuation. See graphics plane.

character string

A grouping of ASCII characters.

cold start

The power-up activities of the Controller. These include clearing all memory including E-Disk, performing a self-test, and loading the operating system. A cold-start occurs when the system is powered up, or when RESTART and ABORT are pressed simultaneously.

Command file

A file that, when designated active by FDOS, is used as a substitute for keyboard inputs. In Fluke Instrument Controllers, a command file with the name STRTUP.CMD is processed each time the Controller is initialized by a cold start or power-on.

Command mode

An IEEE-488 term indicating that a controller has set the ATN (attention) line. In this mode, instruments on the bus are addressed or unaddressed as talkers and listeners.

constants

Fixed values which may be floating-point, integer, or string data types.

control character

Used to produce specific actions such as terminating program execution, exiting from the Editor, halting and restarting scrolling.

controller

A device connected to a bus capable of designating instruments as talkers or listeners by using bus message sequences. A device does not need to be programmable to act as a controller. However, only a controller can examine the data or status of instruments to determine appropriate conditions for designation changes. There can be only one active controller on a bus at one time.

CPU

Central processing unit, the controlling instruction and data processor in any computer system. In the Fluke Instrument Controller, the CPU is the microprocessor and its supporting components located on the Single Board Computer module.

CRT

Cathode Ray Tube, the display screen on the Instrument Controller front panel.

current position

The pixel location defined by X,Y coordinates in the graphics plane that is the starting position for turning the beam on or off to paint a line, or to move to another location. On power up, the current location is X0, Y0.

cursor

The visible pointer on the CRT display that allows the user to recognize the position being pointed to by the system software.

data

Numerical information that has been collected for interpretation by a program.

data base

A stored and defined collection of data that is made available for report generation or further calculations by a program.

Data Base Management System (DBMS)

Any systematic approach to storing, updating, and retrieving information as a common data base for many users.

data file

A file holding either random or sequential access information. Contrasted to a program file.

Data lines

Eight of the sixteen bus lines which carry either data or multiline bus messages (Universal, Addressed and Address commands).

Data mode

The default mode of the bus when the controller has left the ATN (attention) line false. All transfers of data or instructions are between instruments.

data processing

The ability to perform calculations on collected data and formatting it into readable reports.

debugging

Any method of detecting and correcting syntax and structure errors in a program.

default

That option which system software selects when the user does not specify an option.

device

A hardware resource that can act as a source or destination of data. In this manual, device is used in two different ways: 1. To represent the internal devices recognized by the Operating System. In this usage, the Controller's devices are identified by two letters, a number, and a colon. For example, MF0: identifies the mini-floppy drive. 2. The symbol "@" followed by a number from 0 to 30 represents external devices, such as instruments connected to the IEEE-488 bus. The BASIC language statement "PRINT @ 2" followed by program data would address instrument 2 as a listener device, and send it program data.

Device Address

A number used by a program to designate an external device for data transfer.

Display control

An ANSI-standard character sequence of ASCII characters which produces a desired display effect such as cursor position or reverse image.

E-Disk

Fluke Trade Mark for the Electronic disk, a memory configuration that makes use of memory as if it were a file-structured device. See Electronic Disk.

editor

A system software program that enables a user to generate and update an application program.

EIA

Electronic Industries Association, publishers of standard RS-232-C for serial data ports.

Electronic Disk

A portion of the memory designated as a file-structured device. Part of the system's dynamic RAM memory is configured so that it functionally emulates a floppy disk. The electronic disk is about 100 times faster than the floppy disk and has no moving parts to wear or cause noise.

EPROM

Erasable Programmable Read Only Memory. A ROM that can be erased and reprogrammed by an equipment manufacturer using specialized equipment.

Escape sequence

A string of characters including an escape (ESC) character, a numeric parameter and a function code which is recognized as a Display control.

expression

A combination of data-names, numeric literals, and named constants, joined by one or more arithmetic operators in such a way that the expression as a whole can be reduced to a single numeric value.

Extended Listener

A listener instrument that requires a two-byte address. See secondary commands.

Extended Talker

A talker instrument that requires a two-byte address. See secondary commands.

FDOS

Floppy Disk Operating System program. FDOS is the executive monitor program of the 1722A Instrument Controller, and is supplied as a file on the System Disk with the filename FDOS2.SYS. Usually called "the Operating System", FDOS is the Controller's central program. When any other program is exited, FDOS takes control (unless the BASIC statement SET SHELL has been used to change the environment). The purpose of FDOS is to load other programs.

file

A collection of related information designated by name as a unit.

file-structured device

Any bulk memory device where programs and data may be stored and retrieved via a system directory.

File Utility Program (FUP)

The file management program provided with the standard Fluke Instrument Controller software package. Provided on the system disk as a file with the name FUP.FD2, this utility program permits directory listing, transferring, deleting, and renaming files, and formatting, packing, and zeroing the Controller's devices.

firmware

Computer programs and data that are recorded in permanent memory. See ROM.

flag

A symbol that indicates a status condition. System flags can be used to indicate the presence of command files or to indicate a state of system readiness.

floating-point variable

A representation of a general-purpose number. They are characterized by wide range (up to 308 places from decimal) and high resolution (up to 15 places). When displayed without modification, up to seven of the digits are displayed, with the last one rounded if necessary. If the decimal is out of range of the display, an exponent of ten is included to bring it to just left of the first number. For example, .00123456789 is displayed as 0.1234567e-02, and 1234567.89 is displayed as 1234568. Note that the inexactness of floating-point representation occasionally must be considered. For example, IF 7*(1.7)=1 will evaluate false. See integer.

floppy disk

A bulk storage recording device that uses a flexible mylar disk similar to recording tape to record programs and data. The location of information on the disk is identified by track (distance from center) and sector (pie-shaped radial subdivision).

flowchart

A pictorial, symbolic representation of a program. Various shapes represent commands, computations, or decisions. A flowchart is the recommended step between an algorithm specification and program writing. It facilitates understanding and debugging because it breaks the program down into logical, sequential modules.

FUP

See File Utility Program

graphics plane

The portion of the display memory where the lines and patterns displayed by the graphics routines are stored. The area is measured in pixels, rather than bytes. One pixel is the smallest amount of graphics information that can be stored or displayed. The graphics plane is 2048 pixels long and 256 high. The display provides a moveable window looking into the graphics plane. The window is 640 by 224 pixels. See character plane.

handshaking

Refers to the 3-wire hardware protocol used to exchange data on the bus. The three bus lines (DAV, NRFD, and NDAC) indicate a remote instrument's readiness to send or receive data.

hexadecimal

A number system based on 16 digits. Sometimes called hex, the system uses A, B, C, D, E, and F, to represent the numbers above 9.

high-level language

Any programming language that requires conversion through a compiler or interpreter into machine code instructions. Examples of high-level languages are BASIC and FORTRAN.

IEEE

Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY, 10017. The IEEE is the publisher of Standard 488-1978 used for interconnecting instruments to the Fluke Instrument Controller through the bus.

IEEE-488-1980

A bus standard agreed upon by participating instrument manufacturers for the interconnections of instruments into a functional system. Also known as the GPIB (General Purpose Instrumentation Bus). The standard is published and maintained by the IEEE.

Immediate mode

A method to use BASIC directly as each line is typed in rather than storing a sequence of lines as a program for later execution. In Immediate Mode, line numbers are not used and each line is executed as soon as the RETURN key is pressed.

Instrument Controller

In an IEEE-488 system, designates the piece of equipment that asserts control over the bus, and which establishes the roles of other connected euipment as listeners or talkers.

integer variable

A representation of an exact number. They are characterized by limited range (-32768 to 32767) and numeric resolution. Integers are normally used for event counting, and for comparisions where exactness is required. See also floating-point.

interface

A hardware and software connection of a device to a system. For example, in the Fluke Instrument Controller, the DMA/Floppy Interface is needed for the system to gain access to the floppy disk.

interpreter

A system software program that interprets the statements of a high-level language program (such as BASIC), producing and executing machine code.

lexical file

An intermediate form of an application program that occupies less space and eliminates some processing steps for the Fluke BASIC Interpreter. Line numbers are represented in binary format and all commands and operators are reduced to binary form. Lexical files always have ".BAL" extensions.

listener

A bus device designated by a controller to receive data or instructions from a designated talker or controller. There can be more than one listener on a bus at the same time.

loader

A program which places another program into main memory for execution.

logical expression

An expression containing variables, constants, function references, etc., separated by logical operators and parentheses.

logic operator

A function that performs comparisons, selections, matching, etc. In BASIC, the logical operators are AND, OR, NOT, and XOR. These are used for either Boolean operation or for bit-manipulation.

machine code

The coded bit-patterns of directly executable machine-dependent computer instructions, represented by numbers or binary patterns.

machine-dependent program

A program that operates on a particular model of computer.

machine-independent program

A program that operates on any computer system that has the necessary hardware and supporting software.

main memory

The RAM memory from which the microcomputer directly executes all instructions and which is used for fast, intermediate storage of data or programs.

main memory array

An array that is stored in main memory.

management lines

Five of the sixteen lines on an IEEE-488 bus. The lines are ATN (Attention), IFC (Interface Clear), REN (Remote Enable), EOI (End Or Identify), and SRQ (Service Request), and call for an immediate and specific action, or flag a condition existing on the bus.

Operating System

A computer program that manages the resources of computer through task scheduling, I/O handling, and file management. See FDOS.

operator

A term for symbols within an application program (such as + or <) that identify operations to be performed.

Operator's Keyboard

The Touch-Sensitive Display.

parallel poll

A method of simultaneously checking the status of up to eight instruments on a bus by assigning each instrument a data line to transmit a service request.

parity

A method of error detection that uses one extra bit for each unit of information (such as a byte). The parity bit is set to one or zero so that the total number of one-bits in the byte is even or odd.

pathname

The full designation of a file. The three parts are the device name, file name and extension. The first two are separated by a colon, and the last two by a period.

pixel

Acronym for picture element; the smallest amount of visual information that the display is able to resolve; one dot.

port

A connection point used for data transfer. See interface.

Primary command

An ASCII character typically used as a bus command.

program

Any meaningful sequence of computer instructions that cause a system to accomplish a desired task.

PROM

Programmable Read Only Memory, a memory IC that can be recorded by an equipment manufacturer using specialized equipment.

protection state

Files prepared on the Controller are assigned a value, either + or – to indicate the intent of the author either to prevent or allow alteration. A file with the + state is protected and will not be written over. A – state indicates that the file may be altered if desired. All newly created files are assigned the – state. All files supplied on the System disk with the Controller are protected. The File Utiltiy Program includes commands for changing the protection state of files.

protocol

A set of rules for exchange of information between a system and a device or between two systems.

RAM

Random Access Memory. Through common usage, the term has come to mean the high-speed volatile semiconductor memory that is normally used for system and user memory.

random access

A method of obtaining information out of memory; each word of a file can be accessed via its own discrete address. See also sequential access.

raster

The scanning pattern of an electron beam on a CRT display. A raster display uses the same scan pattern all the time, forming images by turning the beam on and off at appropriate times.

RESTART

Front panel switch that resets the system. When pressed alone, RESTART causes a warm start. When pressed at the same time as ABORT, causes the system to perform a cold start.

ROM

Read Only Memory, used for permanently recorded computer programs and data.

RS-232-C

A digital communications standard agreed upon by participating manufacturers of data communication equipment for the transfer of serial digital data between data communication equipment (DCE) and data terminal equipment (DTE). The 1722A is a DTE device. The standard is published and maintained by the Electronic Industries Association.

scientific notation

A system for describing real or integer numbers via a shorthand form of floating-point notation.

Secondary command

IEEE-488 bus commands used to increase the address length of extended talkers and listeners to two bytes.

serial data

Information transmitted one bit at a time over a single wire at a predefined baud rate.

sequential access

A method of accessing data in a file by looking at each piece of data, in order, until a match is found. See also random access.

serial poll

A method of sequentially determining which instrument on a bus has requested service. One instrument at a time is checked via the eight data lines.

serial port

An external connector that conforms to the industry standard RS-232-C. Normally, asynchronous ASCII codes are used unless otherwise desired.

SET Utility Program

The program that changes the parameters of the 1722A's serial communications ports. Supplied on the System Disk with the filename SET.FD2, this program permits configuring the Controller so it is able to communicate with other devices that implement the RS-232 Serial Data Communications standard. Parameters that can be changed include baud rate, parity bit, number of bits per character, stall input and output characters, and time out value.

shell

The Controller's environment, either defaulted to FDOS or changed by the BASIC language SET SHELL statement. When a program finishes, the Controller returns to the program named by the SET SHELL statement, rather than to the FDOS prompt.

simple variable

Fluke BASIC program variable that is either an integer or floating-point value (not a character string) and contains only one value (not dimensional).

soft-sectored

In floppy disks, the beginning of every sector on a disk is determined by checking certain data patterns. Hard-sectored disks have predetermined sector beginnings designated by a physical marker, such as a hole.

software

Computer programs and data recorded and used on a medium that can be erased and rewritten by program command.

source

This term has two meanings: 1. The pathname where information presently resides when using a File Utility Program command that moves a file from one place to another; the input side of the channel. 2. An instrument connected to the bus and transmitting either command mode or data mode information.

string variable

An expression that represents collections of characters that may or may not be numeric.

structured programming

A method of programming which require an initial design process to lay out the program structure in a modular form. Structured programming minimizes 'spaghetti code' programs by keeping GOTO statements to a minimum and by using subroutines to structure the program into discrete, easily readable modules.

subroutine

A section of a program that performs a specific function on request of the main program or another subroutine. Subroutines are used in BASIC via the GOSUB statement.

synchronous data

Digital information transmitted in predetermined message block sizes with a clock signal to synchronize the receiver. See asynchronous data.

syntax

The proper grammar required for an interpreter to recognize and execute a program statement.

syntax diagram

A pictorial representation of the grammar required for the execution of a program statement.

system

Any interconnection of instruments or other devices that cooperate to accomplish a task. A controller is an essential part of a system whenever the designations of talkers and listeners needs to be changed during the task. A controller is a necessary part of any system that requires data processing or a centralized control point.

System Device

The designated file-structured device on the Controller that acts as the primary file storage module. The floppy disk or electronic disk may be designated as the system device by the File Utility program's Assign option. The floppy disk drive (MF0:) is the default system device.

system directory

The listing of program and data files on a bulk storage, file structured device.

system memory

Those portions of the Random Access Memory allocated for use by the operating system and utilities or BASIC Interpreter.

system software

The collection of programs used to handle file management procedures on a system.

Talker

An IEEE-488 connected instrument that has been designated by the controller on the bus to send data to listeners.

Time and Date Utility Program

The program that sets the time and date of the Controller's real time clock. Supplied on the System Disk with the filename TIME.FD2, this program accepts the time and date by keyboard inputs, and transmits the information to the real-time clock. With battery back-up, the clock maintains the correct time and takes into account leap years. The clock can be used to time and date stamp programs or data collected by programs, or to perform an operation at a specified time.

Touch-Sensitive Display

The combination of the display screen and the touch-sensitive panel which acts as the operator's keyboard.

warm start

The activities the Controller performs when the RESTART switch is pressed: ceases the current operation and reloads (reboots) the Operating System. See cold start.

Universal command

A message sent across the data lines of a bus that affects all connected instruments whether or not they are designated as listeners.

user memory

Area reserved in main memory for storage and execution of user-written application programs and data.

variable

A representation of a quantity, or the quantity itself, which can assume any of a given set of values. A variable may be integer, string, or floating point value designators.

virtual array

A matrix stored on a file-structured storage medium as a random access file. Virtual arrays can be integer, string, or floating-point arrays with one or two dimensions. Once a virtual array file has been opened and the virtual array has been dimensioned, the array elements are handled by the programmer exactly as they are in main memory array.

yank buffer

A temporary memory location where the System Editor program can store data "yanked" from a file.

Appendix N
Reserved Words

Table N-1 lists reserved words that conflict with BASIC language statements. These reserved words may NOT be used in Interpreted BASIC as

- Variable names
- Statement label names
- Literal subroutine names (in most cases)

  The reserved words may be used as subroutine names with the following restrictions:

- Implied CALL statements may not be used with subroutines using a reserved name. For example, the implied CALL statement to a subroutine named STOP

  ```
  STOP
  ```

  will execute a BASIC STOP function. Use the statement

  ```
  CALL STOP
  ```

  to branch to the subroutine.

- The name "FN" may not be used for any purpose other than a user-defined function. All strings beginning with FN are considered functions.

Table N-1. Reserved Words

ABS	ERROR	OLD	SQR
ALL	EXEC	ON	SRQ
AND	EXIT	OPEN	STEP
AS	EXP	OR	STIME$
ASCII	FILE	PACK	STOP
ASH	FLEN	PASSCONTROL	TAB
ASSIGN	FN	PI	TAN
ATN	FOR	PORT	TERM
BREAK	GOSUB	PORTSTATUS	THEN
CALL	GOTO	PPL	TIME
CHR$	IF	PPOL	TIME$
CLEAR	INCHAR	PPORT	TIMEOUT
CLOCK	INCOUNT	PRINT	TO
CLOSE	INIT	PROTECT	TRACE
CMDFILE	INPUT	QDIR	TRIG
CMDLINE$	INSTR	RAD$	TRIM
COM	INT	RANDOMIZE	UCASE$
CONFIG	INTERVAL	RBIN	UNLINK
CONT	KEY	RBYTE	UNPROTECT
COPY	KILL	READ	USING
COS	LCASE$	REM	VAL
CPOS	LEFT	REMOTE	WAIT
CTRL/C	LEN	REN	WBIN
DATA	LET	RENAME	WBYTE
DATE	LINE	RESAVE	WITH
DATE$	LINK	RESAVEL	XOR
DEF	LIST	RESTORE	
DELETE	LN	RESUME	
DIM	LOCAL	RETURN	
DIR	LOCKOUT	RIGHT	
DISABLE	LOG	RND	
DUPL$	LSH	RUN	
ECHO	MEM	SAVE	
EDIR	MID	SAVEL	
EDIT	MOD	SET	
ELSE	NEW	SGN	
ENABLE	NEXT	SHELL	
END	NOECHO	SIN	
ERL	NOT	SIZE	
ERR	NUM$	SPACE$	
	OFF	SPL	

INDEX

r refers to a statement number in the BASIC Reference manual.

Abbreviations, **M-1**
ABS, **r1, 8-5**
Alternate Character set, **14-6**
AND, **r69, 6-18**
ANSI Standards, **2-4**
  Exceptions to, **2-4**
  Mode, **14-13**
Arithmetic Operators, **r-2, 6-12**
Arrays
  Advantages and Disadvantages, **7-6**
  CLOSE, **r13, 7-8**
  Dimensioning, **7-13**
  Element Access, **7-35**
  Equivalent Virtual Arrays, **7-38**
  I/O Channels, **7-7, 7-8**
  In Chained Programs, **7-9, 13-9, 13-10**
  Main Memory Arrays, **7-13**
    DIM, **r24, 7-14**
    using, **7-14**
    programming techniques, **7-15**
    storing, **7-19**
      device size requirements, **7-20**
      device size calculation, **7-21**
    two dimensional arrays, **7-16**
  Multiple Arrays, **7-17**
  OPEN, **r99, 7-8**
  Organization, **7-27**
  Programming Techniques, **7-15**
  Redimensioning, **7-17**
  Reusing Declarations, **7-37**
  size, **7-9**
  TRACE ON array, **15-8**
  Types and Differences, **7-5**
  Two-Dimensional Arrays, **7-16**
  Using, **7-4**
    Strings, **7-34**
    Main Memory Arrays, **7-14**
    Main Memory Arrays as variables, **7-14**
  Variables, **6-9**
  Virtual Arrays, **7-22**
    Advantages and Disadvantages, **7-24**
    Array Element Access, **7-35**
    Chaining Techniques, **13-9**
    Creating, **7-32**
    Definition, **7-24**
    DIM, **r24, 7-14**
    Equivalent Virtual Arrays, **7-38**
    file structure discussion, **7-26**
    File Organization, **7-27**
    Opening array file, **7-22**
    Programming Techniques, **7-35**
    Reusing declarations, **7-37**
    Size Requirements, **7-29**
    Size Calculation, **7-30**
    Splitting Among Files, **7-36**

Using, 7-33
as Ordinary Variables, 7-33
Virtual Array Strings, 7-34
Array Variables, r3
ASCII, r4, 8-11
ASCII Character Set, 14-5, G-1
ASH, r5, 8-5
Assembly Language
Error handler, H-1
Subroutines, 12-1, H-1
ASSIGN, r6, 3-7
Assignment Operator, r7
Assignment Statement (LET), r62, 6-12
ATN, r8, 8-9

BACK SPACE Key, 4-20
BASIC
Conventions, 2-1
Enhancements, 1-2
Editor, 4-17
Exceptions to ANSI, 2-4
Exiting, 2-6
Operating Modes, 2-7
Reference Textbook, 1-4
Running BASIC, 2-5
Variables, 6-8
Binary Numbers, 6-16
Twos Complement, 6-18
BREAK, r9, 10-14

CALL, r10, 12-5
Chaining, 13-1
Virtual Arrays, 13-9
Changing the Sequence of Program Lines, 4-14
channel I/O, 7-7
Character
Attributes, 14-25
Graphics, 14-28, F-1
CHR$, r11, 8-11, 14-9
CLEAR, r12, 9-9, 9-10
CLOSE, r13, 7-8
CMDFILE, r14
CMDLINE$, r15, 13-6
COM, r16, 13-8
Common Memory Requirements, A-1
Conditional Expressions,
r41, r42, r118, r126, 11-8, 12-4
CONFIG, r17, 9-21
CONT TO, r18, 15-15
COPY, r19, 3-5, 7-10
COS, r20, 8-9
CPOS, r21, 8-11, 14-10
⟨CTRL /Key-Modifier, 4-4
⟨CTRL⟩/C, 4-4
⟨CTRL⟩/F, 4-5
⟨CTRL⟩/P, 4-4
⟨CTRL⟩/Q, 4-5
⟨CTRL⟩/R, 4-4
⟨CTRL⟩/S, 4-5
⟨CTRL⟩/T, 4-5
⟨CTRL⟩/U, 4-5

DATA
statement, r117
storage, 7-1
using arrays for, 7-4
types, 6-4, r38, r56, r147
DATE$, r150, 6-11
Debugging, 15-1
DEF FN, r22, 8-14
DELETE, r23, 4-10
Delete file (KILL), r58, 3-6
DELETE Key, 4-3, 4-20
DELETE CHAR Key, 4-20
DELETE LINE Key, 4-20
Device Management Functions, 3-7
ASSIGN,r6, 3-7
PACK, r100, 3-6
PROTECT, r109, 3-7
UNPROTECT, r161, 3-7
Devices, 9-5, M-1
DIM, r24, 13, 7-27
DIR, r25, 3-5
DISABLE, r26, 11-30

disable keyboard, **14-32**
DISABLE CMDFILE, **r27**
Display
  ANSI Compatible Control Sequences, **14-14**
  Character
    Attributes, **14-20, E-1**
  Graphics, **14-28, F-1**
    size, **14-27**
  Control Characters, **14-12**
    summary, **E-1**
    sequences, **14-13, 14-16, E-1**
  Cursor Control, **14-17**
  Erasing, **14-18**
  Field Attributes, **14-22**
  Graphics, **K-1, F-1**
  Input, **14-33**
    KEY variable, **r57, 14-34**
  Mode Commands, **14-19, 14-14**
  Non-destructive character, **14-26**
  Output, **14-4**
  Scrolling, **14-17**
  Visual Attributes, **14-20**
DUPL$, **r28, 8-11**

EDIR, **r29, 3-5**
EDIT, **r30, 4-15, 4-17**
Edit Mode, **2-9**
  Keys, **4-19**
editing
  general, **2-9, 4-15**
  using edit mode, **4-17, 4-19, 4-22**
  using the system editor, **4-15**
ENABLE, **r31, 11-30**
ENABLE CMDFILE, **r32**
END, **r33**
Entering a Program, **4-7**
  Immediate Mode, **4-7**
Erase display screen, **E-1, 14-18**
ERL, **r150, 6-11**
ERR, **r150, 6-11**
Error
  Handler, Assembly Language, **H-1**
  Interrupt, **r89, 11-4**
  Levels, **11-6, 11-28**
  messages, **16-1**
  Variables, **r150, 6-11, 11-29**
EXEC, **r34, 13-6**
EXIT, **r35, 2-6**
Exiting BASIC, **r35, r138, 2-6**
EXP, **r36, 8-5**
Expressions, **6-12**

F$RGMY, **12-10**
Fatal Errors, **11-4, 11-8**
Field Attributes, **14-20, 14-22**
File Management
  System Functions for, **3-4**
File Names, **1-7, 2-11**
files
  random access, **7-6, 7-22**
  sequential, **7-6, 13-12**
    reading, **7-11**
  Virtual Array, **7-27**
FLEN, **r37, 6-11**
Floating-Point, **r38**
  Constants, **6-5**
  Data, **6-4**
  Variables, **6-8**
FOR - NEXT, **r39**
Formatted Printing, **r108, 14-7**
FORTRAN Subroutines, **I-1**
Functions, **8-1**
  Math functions, **r71, s8-4**
  String functions, **8-10**
  Trig functions, **r156, 8-8**
  User Defined, **r22, 8-14**

GOSUB, **r40, 12-4**
GOTO, **r41**
Graphics, **F-1, K-1**

IEEE-488 Bus
  Addressing, **9-4**
    Devices, **9-5**

Port, 9-8
Serial IEEE-488 devices, 9-7
Bus Codes, B-1
Bus Management Lines, B-1
Bus Messages, B-1
CLEAR, r12, 9-9, 9-10
CONFIG, r17, 9-21
Command Message Sequences, B-1
Control statements, 9-9
Data Lines, B-1
Data Transfer Statements, 9-17
Device addressing, 9-5
Handshake Lines, B-1
INIT, r45, 9-10
Initialization and Control, 9-9
Input and Output Statements, 9-14
INPUT, r46, 9-14
I/O, 9-14
LOCAL, r66, 9-11
LOCKOUT, r67, 9-11
OFF PPOL, r83, 9-21
OFF SRQ, r85, 9-21
ON PORT, r94, 9-12
ON PPOL, r95, 9-21
ON SRQ, r97, 9-21
PASSCONTROL, r101, 9-12
Polling Statements, 9-20
Port addressing, r102a, 9-8
PORTSTATUS, r103, 9-22
PPL, r104, 9-22
primary address, 9-4
PRINT, r105, r106, r107, 9-16
PRINT USING, r108, 9-16
RBIN, r113, 9-19
RBIN WBYTE, r114, 9-19
RBYTE, r115, 9-18
RBYTE WBYTE, r116, 9-18
REMOTE, r120, 9-12
secondary address, 9-5
SET SRQ, r138, 9-13
Standard, 9-3
SPL, r143, 9-22
TERM, r153, 9-13
TIMEOUT, r154, 9-13
TRIG, r157, 9-13
WAIT, r163, 9-23
WAIT FOR TIME, r164, 9-23
WBIN, r168, 9-20
WBYTE, r169, s9-18
IF conditional statement
IF - GOTO, r42
IF - THEN, r42
IF - THEN - ELSE, r42
Immediate Mode, 2-7
INCHAR, r43, 10-13
INCOUNT, r44, 10-12
INIT, r45, 9-9
INPUT, r46, r47, 9-14
LINE, r49, r50, r51, r52, 9-15
LINE #n, r50
LINE @, r51
LINE WBYTE, r52, 9-15
WBYTE, r53, 9-15
#n, r48, 10-9
@, r47, 9-4
Input and Output Statements, r46, r105
IEEE-488, 9-14
INSTR, r54, 8-12
INT, r55, 8-5
Integer
constants, r56, 6-6
data, r56, 6-5
variables, r56, 6-8
Internal Data Format, 12-14
Internal Structure of Variables, A-1
Interrupt
Control Statements, 11-30
Error Handling, 11-28
Error Variables, 11-29
Errors
Fatal, 11-28
Recoverable, 11-28
Warning, 11-29
Hierarchy, 11-6

OFF EVENT Interrupts
CLOCK, **r78, 11-19**
CTRL/C, **r76, 11-10**
ERROR, **r79, 11-7**
INTERVAL, **r80, 11-20**
KEY, **r81, 11-15**
PORT, **r82, 11-15**
PPOL, **r83, 11-19**
PPORT, **r84, 11-15**
SRQ, **r85, 11-11**
#n, **r77, 11-13**
ON-EVENT Interrupts, **11-7**
CLOCK, **r87, 11-19**
CTRL/C, **r88, 11-10**
ERROR, **r89, 11-8**
INTERVAL, **r92, 11-20**
KEY, **r93, 11-14**
PORT, **r94, 11-15**
PPOL, **r95, 11-18**
PPORT, **r96, 11-15**
SRQ, **r97, 11-16**
#n, **r90, 11-12**
Processing, **11-1**
program examples, **11-31**
RESUME, **r125, 11-21**
Types, **11-4**
CLOCK, **11-5**
Error, **11-4**
KEY, **11-5**
INTERVAL, **11-5**
PORT, **11-5**
PPOL, **11-5**
PPORT, **11-5**
SRQ, **11-5**
CTRL /C, **11-4**
#n, **11-5**
WAIT Interrupts, **r163, r164, 11-22, 11-23**
WAIT time, **r163, 11-24**
WAIT FOR KEY, **r165, 11-25**
WAIT FOR PPOL, **r166, 11-27**
WAIT FOR SRQ, **r167, 11-26**
I/O Buffer Memory Requirements, **A-3**
KEY Interrupt, **r165, 11-5, 11-14**
KEY Variable, **r57, 14-34**
Keyboard enable and disable, **14-32, E-1**
KILL, **r58, 3-6**

LCASE$, **r59, 8-12**
LEFT, **r60, 8-12**
LEN, **r61, 8-12**
LET, **r62, r7, 6-12**
LINE FEED Key, **4-21**
line numbers, changing, **4-11, r121, r30**
LINK, **r63, 12-5**
LIST, **r64, 4-9**
LN, **r65**
LOCAL, **r66, 9-11**
LOCKOUT, **r67, 9-11**
LOG, **r68, 8-6**
Logical Operators, **r69, 6-16**
Hierarchy, **6-20**
Loops, **r39**
LSH, **r70, 8-6**

Main Memory Arrays, **7-13**
using, **7-14**
programming techniques, **7-15**
storing, **7-19**
device size requirements, **7-20**
device size calculation, **7-20**
two dimensional arrays, **7-16**
Mathematical Functions, **r71, 8-4**
MEM, **r72, 6-11**
Memory requirements
common area, **A-4**
for I/O buffer, **A-3**
for variables, **A-1**
MID, **r73, 8-12**
Mnemonics list, **M-1**
MOD, **r74, 8-6**
Modes, **2-7**
display, **14-13, 14-14**
multiple statements (\), **2-4, 2-7, 4-8**

NEXT, **r39**
NEXT PAGE Key, **4-3**
NOT, **r69, 6-20**
Non-destructive character, **14-26, E-1**
NUM$, **r75, 8-13**
Notation Conventions, **1-9**
Numeric Comparisons, **6-16**

OFF event
  OFF CLOCK, **r78, 11-19**
  OFF ⟨CTRL⟩/C, **r76, 11-11**
  OFF ERROR, **r79, 11-9**
  OFF INTERVAL, **r80, 11-20**
  OFF KEY, **r81, 14-25, 11-15, 14-35**
  OFF PORT, **r82, 11-15**
  OFF PPOL, **r83, 11-19**
  OFF PPORT, **r84, 11-15**
  OFF SRQ, **r85, 11-17**
  OFF #n, **r77, 11-13**

OLD, **r86, 5-6**
ON event
  CLOCK, **r87, 11-19**
  ⟨CTRL⟩/C GOTO, **r88, 11-10**
  Interrupts, **11-4**
  ON ERROR GOTO, **r89, 11-8**
  ON GOTO, **r91**
  ON #n GOTO, **r90, 11-12**
  ON GOSUB, **r91**
  ON INTERVAL GOTO, **r92, 11-20**
  ON KEY GOTO, **r93, 11-14**
  ON PORT GOTO, **r94, 11-15**
  ON PPOL GOTO, **r95, 11-18**
  ON PPORT GOTO, **r96, 11-15**
  ON SRQ GOTO, **r97, 11-16**

OPEN, **r98, r99**
  AS NEW, **7-8**
  AS OLD, **7-8**
  virtual array, **7-32**
Operating Modes, **2-7**
Operators and Expressions, **6-12**
Operator Hierarchy, **6-20**

OR, **r69, 6-19**
Output and Input (RS-232), **10-9**

PACK, **r100, 3-7**
PAGE Mode Switch, **4-3**
Parallel Poll Enable Codes, **D-1**
Parameter Decoding Subroutine **F$RGMY, 12-10**
Parameter Passing Mechanism, **12-9**
Parentheses (use), **6-22**
PASSCONTROL, **r101, 9-12**
PI (3.1417...), **r102, 6-7**
Polling statements, **9-20**
PPL, **r104, 9-22**
PPOL Interrupt, **r95, 9-21**
PRINT, **r105, 10-9, 14-7**
PRINT #n, **r107, 10-9**
PRINT @, **r106**
PRINT USING, **9-14, 9-16, 10-9, 14-8**
Program Chaining
  and main memory arrays, **7-18**
  chaining, **13-1**
  Statement Definitions, **13-4**
  Using Sequential Files, **13-12**
program examples
  alternate TAB function, **14-9**
  array chaining, **13-10**
  CPOS statement, **14-10**
  character graphics, **14-28**
  create a sequential file, **7-10**
  display demo, **14-31**
  display graphics demo, **K-3**
  using DEF FN, **8-14**
  error handling, **11-31**
    CTRL/C handler, **11-35**
    using bus timeout, **11-32**
    using ERR and ERL, **11-29**
    using RESUME, **11-21, 11-33**
  GOSUB statement, **r40, 12-4**
  interactive display program, **14-39**
  interrupt processing, **11-31**
  LOAD ARRAY, **7-4**
  open output chan. to virtual array, **7-8**
  open input chan. to virtual array, **7-8**

read IEEE-488 bus, **9-4**
read sequential file, **7-11**
TSO usage, **14-33**
virtual array dimensioning, **J-1**
visual attributes, **14-20**
WAIT statement, **11-36**

Program Debugging, **15-1**
CONT TO, **r18, 15-15**
STEP, **r144a, 15-14**
STOP ON, **r149, 15-13**
TRACE ON/OFF, **r155, r154A, 15-4, 15-13**
trace by line number, **15-5**
Variable tracing, **15-8**
Programming, **1-3**
PROTECT, **r109, 3-7**

QDIR, **r110, 3-6**

Radian Angular Measure, **r156, 8-8**
RAD$, **r111, 8-13**
RANDOMIZE, **r112, 8-7**
RBIN, **r113, 9-19**
RBIN WBYTE, **r114, 9-19**
RBYTE, **r115, 9-18**
RBYTE WBYTE, **r116, 9-18**
READ, **r117**
Recoverable Errors, **11-28**
Relational Operators, **r118, 6-14**
REM (!), **r119, 4-6**
REMOTE, **r120, 9-12**
REN, **r121, 4-11**
RENAME, **r122, 3-6**
RESAVE, **r124, 5-4**
RESAVEL, **r124, 5-4**
Reserved words, **N-1**
RESTORE, **r117**
RESUME, **r125, 11-7, 11-21**
RETURN, **r126, 12-4**
Reverse video (display), **E-1, 14-20**
RIGHT, **r127, 8-13**
RND, **r128, 8-7**

RS-232 Serial I/O, **10-1**
BREAK, **r9, 10-14**
CLOSE, **r13, 10-8**
Defined, **10-4**
Device and Port Addressing, **10-5**
echo mode, **r134, r136, 10-12**
Establishing Serial Communications, **10-15**
INCHAR, **r43, 10-13**
INCOUNT, **r44, 10-12**
INPUT, **r46, r48, 10-10**
I/O Channels, **10-6**
noecho mode, **10-12**
OPEN, **r98, 10-7**
Output and Input, **10-9**
PRINT, **r105, r107, 10-9, 14-7**
PRINT USING, **r108, 10-9, 14-8**
RUN, **r129, 5-8, 13-4**
RUN WITH, **r129, 13-6**
Run Mode, **2-9**
Running A Program, **5-6**

Saving A Program, **5-4, r130, r124**
SAVE, **r130, 5-4**
SAVEL, **r130, 5-4**
sequence of line numbers, changing, **4-14**
Sequential Data Files, **7-10, 13-12**
SET CLOCK, **r131, 11-19**
SET CMDLINE$, **r132**
SET DATE, **r133**
SET ECHO, **r134, 10-12**
SET INTERVAL, **r135, 11-20**
SET NOECHO, **r136, 10-12**
SET SHELL, **r138**
SET SRQ, **r137, 9-13**
SET TIME, **r139**
SGN, **r140, 8-7**
SIN, **r141, 8-6**
SPACE$, **r142, 8-13**
Special Function Keys, **4-3**
SPL, **r143, 9-22**
SQR, **r144, 8-7**
SRQ Interrupt, **r137, r97, 11-16, 9-20**
STEP, **r144A, 15-14**

Step Mode, 2-5, 2-10, 15-14
STIME$, r145
STOP, r148
STOP ON, r149, 15-13
String
  Comparisons, r146
  Constants, r147, 6-7
  Conversion to Internal Form, 12-19
  Conversion from Internal Form, 12-20
  Data, 6-6
  Functions, 8-10
  Variables, 6-8
Subroutine(s), 12-1
  assembly language, 12-7
  Format, 12-9
  FORTRAN, 12-8
  External, 12-5
    Software Required, 12-6
  Internal, 12-4
  Linkage Mechanism, 12-10
  Names, 12-6
  Parameter Decoding, 12-10
  Parameter Format, 12-14
  Parameter Passing, 12-9
  parameter syntax, 12-17
Symbol explanation, 1-8
Syntax
  diagrams
    Command Line, 2-8
    Program Line, 2-8
    Remark Line, 2-8
    Program, 2-8
  How to Read, 1-7
  Supplementary, L-1
System
  Constants, 6-7
  Devices, 3-7
  Variables, r150, 6-11

TAB, r151, 8-13, 14-9
TAN, r152, 8-9
TERM, r153, 9-13
TIME, r150, 6-11
TIME$, r150, 6-11
Time expression, M-1, r163, 11-24
TIMEOUT, r154, 9-13
Touch Sensitive Display, 14-1
  KEY System Variable, r57, 14-34
  ON KEY interrupt, r93, 11-14, 14-35
  Using for input, 14-33
  Using for Output, 14-4
  Cursor movement, 14-17
  WAIT FOR KEY, r165, 14-36
TRACE OFF, r154A, 15-13
TRACE ON, r155, 15-4
Trace Options, 15-11
Trigonometric Functions, r156, 8-8
TRIG statement, r157, 9-13
TRIM, r158, 7-34
TSO (Touch Sensitive Overlay), 14-1

UCASE$, r159, 8-14
underlining (display), 14-20, E-1
UNPROTECT, r161, 3-8
User Defined Functions, r22, 8-14

VAL, r162, 8-14
Variable Storage Requirements, A-1
Variable Structure, A-1
Variables, r38, r56, r147, r150, 6-8
  Array, 6-9
  Floating-Point, 6-8
  Integer, 6-8
  Introduction, 6-8
  String, 6-8
  System, 6-11
Virtual Arrays, 7-22
  Advantages and Disadvantages, 7-24
  Array Element Access, 7-35
  Chaining Techniques, 13-9
  Creating, 7-32
  Definition, 7-24
  DIM, 7-22
  Dimensioning Program, J-1
  Equivalent Virtual Arrays, 7-38

file structure discussion, **7-26**
File Organization, **7-27**
Opening array file, **7-32**
Programming Techniques, **7-35**
Reusing declarations, **7-37**
Size Requirements, **7-29**
Size Calculation, **7-30**
Splitting Among Files, **7-36**
Using, **7-33**
as Ordinary Variables, **7-33**
Virtual Array Strings, **7-34**

WAIT, **r163, 9-23, 11-24**
WAIT FOR event, **r164, 11-22**
WAIT FOR KEY, **r165, 11-25**
WAIT FOR PPOL, **r166, 11-27**
WAIT FOR SRQ, **r167, 11-26**
WAIT FOR TIME, **r164, 11-24**

WBIN, **r168, 9-20**
WBYTE, **r169, 9-18**
WBYTE Decimal Equivalents, **C-1**
Writing a Program, **4-6**

XOR, **r69, 6-19**

Symbols
=, **r7, 6-7, 6-8, 6-14**
%, **r56, 6-6**
$, **r147, 6-7**
+, **r2, 6-13, 6-15**
-, **r2, 6-13**
*, **r2, 6-13**
/, **r2, 6-13**
^, **r2, 6-13**
!, **r119**
@, **r106, 9-4**